Self-improving Algorithms for Coordinate-Wise Maxima and Convex Hulls*

Kenneth L. Clarkson† Wolfgang Mulzer‡ C. Seshadhri§

November 6, 2012

Abstract

Computing the coordinate-wise maxima and convex hull of a planar point set are probably the most classic problems in computational geometry. We give an algorithm for these problems in the *self-improving setting*. We have n (unknown) independent distributions $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$ of planar points. An input set of points $(p_1, p_2, \ldots, p_n)$ is generated by taking an independent sample p_i from each $\mathcal{D}_i$, so the input distribution $\mathcal{D}$ is the product $\prod_i \mathcal{D}_i$. A self-improving algorithm repeatedly gets inputs from the distribution $\mathcal{D}$ (which is *a priori* unknown) and tries to optimize its running time for $\mathcal{D}$. Our algorithm uses the first few inputs to learn salient features of the distribution and then becomes an optimal algorithm for distribution $\mathcal{D}$. Let OPT-MAX$_\mathcal{D}$ (resp. OPT-CH$_\mathcal{D}$) denote the expected depth of an *optimal* linear comparison tree computing the maxima (resp. convex hull) for distribution $\mathcal{D}$. Our maxima algorithm eventually has an expected running time of $O(\text{OPT-MAX}_\mathcal{D} + n)$, even though it did not know $\mathcal{D}$ to begin with. Analogously, we have a self-improving algorithm for convex hull computation with expected running time $O(\text{OPT-CH}_\mathcal{D} + n \log\log n)$.

Our results require new tools for manipulating linear comparison trees. In particular, we show how to convert a general linear comparison tree to a very restricted version, which can then be related to the running time of our algorithms. Another interesting feature of our approach is an interleaved search, where we try to determine the likeliest point to be extremal with minimal computation. This allows the running time to be competitive with the optimal algorithm for the distribution $\mathcal{D}$.

1 Introduction

The problems of planar maxima and convex hull computation are classic computational geometry questions and have been studied since at least 1975 [23]. They have well-known $O(n \log n)$ time comparison-based algorithms (where n is the number of points), with matching lower bounds. Further research has addressed a wide variety of more advanced settings: one can achieve linear expected running time for uniformly distributed points in the unit square; output-sensitive algorithms

*Preliminary versions appeared as K. L. Clarkson, W. Mulzer, and C. Seshadhri, *Self-improving Algorithms for Convex Hulls* in Proc. 21st SODA, pp. 1546–1565, 2010; and K. L. Clarkson, W. Mulzer and C. Seshadhri, *Self-improving Algorithms for Coordinate-wise Maxima* in Proc. 28th SoCG, pp. 277–286, 2012.

†IBM Almaden Research Center, San Jose, USA. Email: klclarks@us.ibm.com

‡Institut für Informatik, Freie Universität Berlin, Berlin, Germany. Email: mulzer@inf.fu-berlin.de

§Sandia National Laboratories, Livermore, USA. Email: scomand@sandia.gov

need $O(n \log h)$ time when the output has size h [20]; and there are results for external-memory models [18].

A major problem with worst-case analysis is that it may not reflect the behavior of real-world inputs. Worst-case algorithms are tailor-made for extreme inputs, none of which may occur (reasonably often) in practice. Average-case analysis, on the other hand, tries to address this problem by assuming some fixed distribution on the inputs. For maxima, the property of coordinate-wise independence covers a broad range of inputs, and allows a clean analysis [7], but is unrealistic even so. Thus, the right distribution to analyze remains a point of investigation. Nonetheless, the assumption of randomly distributed inputs is very natural and one worthy of further study.

The self-improving model. Ailon et al. introduced the self-improving model to address this problem with average case analysis [3]. In this model, there is some fixed but unknown input distribution $\mathcal{D}$ that generates independent inputs, that is, whole input sets P. The algorithm initially undergoes a *learning phase*, where it processes inputs with a worst-case guarantee but tries to learn information about $\mathcal{D}$. The aim of the algorithm is to become optimal *for the distribution* $\mathcal{D}$. After seeing some (hopefully small) number of inputs, the algorithm shifts into the *limiting phase*. Now, the algorithm is tuned for $\mathcal{D}$, and the expected running time is (ideally) optimal for $\mathcal{D}$. A self-improving algorithm can be thought of as an algorithm able to attain the optimal average-case running time for all, or at least a large class of, distributions $\mathcal{D}$.

Following earlier self-improving algorithms, we assume that the input has a product distribution. An input $P = (p_1, p_2, \ldots, p_n)$ is a set of n points in the plane. Each p_i is generated independently from a distribution $\mathcal{D}_i$, so the probability distribution of P is the product $\prod_i \mathcal{D}_i$. The $\mathcal{D}_i$s themselves are arbitrary, and the only assumption made is their independence. There are lower bounds [2] showing that some restriction on $\mathcal{D}$ is necessary for a reasonable self-improving algorithm, as we shall explain later.

The first self-improving algorithm was for sorting, and it was later extended to Delaunay triangulations [2, 11]. These results show that *entropy-optimal* performance is achievable in the limiting phase. Later, Bose et al. [5] described *odds-on trees*, which provide a general method to obtain self-improving versions for several query problems, such as planar point location, orthogonal range searching, or point-in-polytope queries.

2 Results

Our main results are self-improving algorithms for planar coordinate-wise maxima and convex hulls over product distributions. Before we can state our theorems formally, we need some basic definitions. First, we explain what it means for algorithms to be optimal for a distribution $\mathcal{D}$. This in turn requires a notion of *certificates*, which allow the correctness of the output to be verified in $O(n)$ time. Any procedure for computing maxima or convex hulls must provide some "reason" to deem an input point p non-maximal/non-extremal. Most current algorithms implicitly give exactly such certificates [17, 21, 23].

Definition 2.1. *Let P be a planar point set. A* maxima certificate *γ for P consists of: (i) the sequence of the indices of the maximal points in P, sorted from left to right; (ii) for each non-maximal point, a* per-point certificate *of non-maximality, which is simply the index of an input point that dominates it. We say that a certificate γ is* valid *for an input P if γ satisfies these conditions for P.*

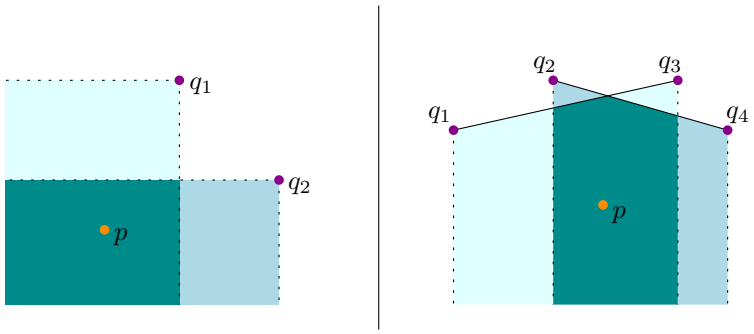

Figure 1: Per-point certificates for maxima and convex hulls. Left: both q_1 and q_2 can be possible per-point certificates of non-maximality for p. Right: Both q_1, q_3 and $q2, q4$ are possible witness pairs for p.

We denote the *upper* convex hull of P by $\text{conv}(P)$. A point $p \in P$ is called *extremal* if it appears on $\text{conv}(P)$, and *non-extremal*, otherwise. For two points $p, q \in P$, we define the *upper semislab* for p and q, $\text{uss}(p, q)$ as the planar region that is bounded by the upward vertical rays through p and q and the line segment $\overline{pq}$. The *lower semislab* for p and q, $\text{lss}(p, q)$ is defined similarly. We call two points $q, r \in P$ a *witness pair* for a non-extremal point p if $p \in \text{lss}(q, r)$.

Definition 2.2. *Given a point set P, a* convex hull certificate *γ for P has: (i) a list of the extremal points in P, sorted from left to right; (ii) a list that contains a witness pair for each non-extremal point in P. Each point in γ is represented by its index in P.*

The model of computation that we use to define optimality is a linear computation tree that generates query lines using the input points. In particular, our model includes the usual CCW-test that forms the basis for many geometric algorithms.

Let ℓ be a directed line. We use ℓ^+ to denote the open halfplane to the left of ℓ and ℓ^- to denote the open halfplane to the right of ℓ.

Definition 2.3. *A* linear comparison tree *$\mathcal{T}$ is a rooted binary tree such that each node v of $\mathcal{T}$ is labeled with a query of the form "$p \in \ell_v^+$?". Here p denotes an input point and ℓ_v denotes a directed line. The line ℓ_v can be obtained in four ways:*

1. *it can be a line independent of the input (but dependent on the node v);*
2. *it can be a line with a slope independent of the input (but dependent on v) passing through a given input point;*
3. *it can be a line through an input point and through a point q_v independent of the input (but dependent on v);*
4. *it can be the line defined by two distinct input points.*

A linear comparison tree is restricted *if it has only only nodes of type (1).*

Let $\mathcal{T}$ be a linear comparison tree and v be a node of $\mathcal{T}$. Note that v corresponds to a region $\mathcal{R}_v \subseteq \mathbb{R}^{2n}$ such that an evaluation of $\mathcal{T}$ on input P reaches v if and only if $P \in \mathcal{R}_v$. If $\mathcal{T}$ is

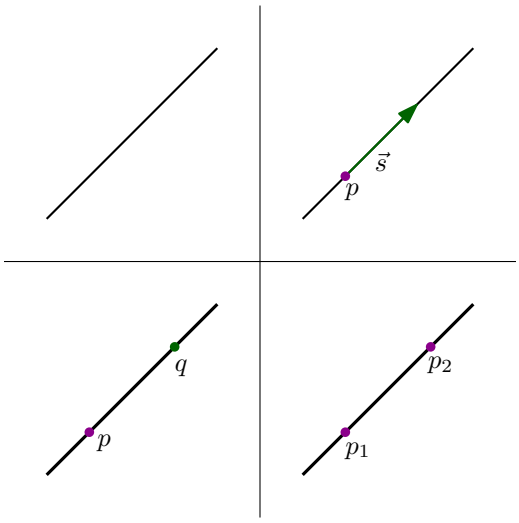

Figure 2: The four types of comparisons: we can compare (1) with a fixed line; (2) a line through an input point p, with a fixed slope $\overrightarrow{s}$; (3) a line through an input point p and a fixed point q; and (4) a line through two input points p_1 and p_2.

restricted, then $\mathcal{R}_v$ is the Cartesian product of a sequence $(R_1, R_2, \ldots, R_n)$ of polygonal regions. Given $\mathcal{T}$, there exists exactly one leaf $v(P)$ that is reached by the evaluation of $\mathcal{T}$ on input P. We now define what is means for a linear comparison tree to compute the maxima or convex hull.

Definition 2.4. *Let $\mathcal{T}$ be a linear comparison tree. If each leaf v of $\mathcal{T}$ is labeled with a maxima certificate that is valid for every input P with $v = v(P)$, we say that $\mathcal{T}$* computes *the maxima of P. The notion that $\mathcal{T}$ computes the convex hull of P is defined analogously.*

The *depth* of node v in $\mathcal{T}$, denoted by d_v, is the length of the path from the root of $\mathcal{T}$ to v. The *expected depth* of $\mathcal{T}$ over $\mathcal{D}$, $d_{\mathcal{D}}(\mathcal{T})$, is defined as $\mathbf{E}_{P\sim\mathcal{D}}[d_{v(P)}]$. Consider some comparison based algorithm A that is modeled by such a tree $\mathcal{T}$. The expected depth of $\mathcal{T}$ is a lower bound on the number of comparisons performed by A.

Let $\mathbf{T}$ be the set of trees that compute the maxima of n points. We define OPT-MAX$_{\mathcal{D}} = \inf_{\mathcal{T}\in\mathbf{T}} d_{\mathcal{D}}(\mathcal{T})$. This is a lower bound on the expected time taken by *any* linear comparison tree to compute the maxima of inputs distributed according to $\mathcal{D}$. We would like our algorithm to have a running time comparable to OPT-MAX$_{\mathcal{D}}$.

Theorem 2.5. *Let $\varepsilon > 0$ be a fixed constant and $\mathcal{D}_1$, $\mathcal{D}_2$, $\ldots$, $\mathcal{D}_n$ be independent planar point distributions. The input distribution is $\mathcal{D} = \prod_i \mathcal{D}_i$. There is a self-improving algorithm to compute the coordinate-wise maxima whose expected time in the limiting phase is $O(\varepsilon^{-1}(n + OPT\text{-}MAX_{\mathcal{D}}))$. The learning phase lasts for $O(n^{\varepsilon})$ inputs and the space requirement is $O(n^{1+\varepsilon})$.*

Analogously, we have a self-improving result for convex hulls. Unfortunately, it is slightly sub-optimal. As before, we set OPT-CH$_{\mathcal{D}} = \inf_{\mathcal{T}\in\mathbf{T}} d_{\mathcal{D}}(\mathcal{T})$, where now $\mathbf{T}$ is the set of trees computing the convex hull of n points. The conference version [9] claimed an optimal result, but it was flawed. Our new analysis is simpler and closer in style to the maxima result.

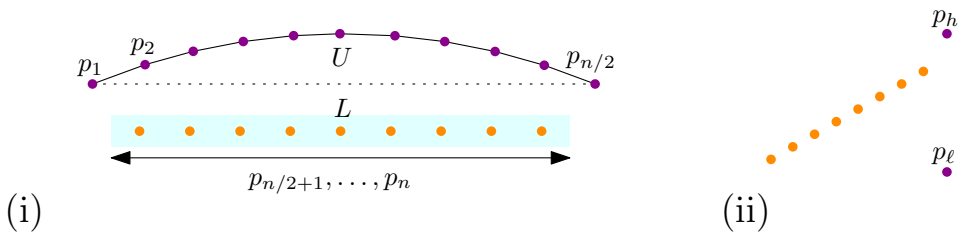

Figure 3: (i) A point set and distribution that is slow in other models: the upper hull U is fixed, while the points $p_{n/2+1}, \ldots, p_n$ roughly constitute a random permutation of the points in L. (ii) Point p_1 is either at p_h or p_ℓ, and its location affects the processing for the other points.

Theorem 2.6. *Let $\varepsilon > 0$ be a fixed constant and $\mathcal{D}_1$, $\mathcal{D}_2$, $\ldots, \mathcal{D}_n$ be independent planar point distributions. The input distribution is $\mathcal{D} = \prod_i \mathcal{D}_i$. There is a self-improving algorithm to compute the convex hull whose expected time in the limiting phase is $O(n \log\log n + \varepsilon^{-1}(n + OPT\text{-}CH_{\mathcal{D}}))$. The learning phase lasts for $O(n^\varepsilon)$ inputs and the space requirement is $O(n^{1+\varepsilon})$.*

Prior Algorithms. With a plethora of planar maxima and convex hull algorithms available, it might seem that there could already be one that is self-improving. A single example shows that for several prior algorithmic approaches, this is not so. We will focus on convex hulls throughout this discussion, though it is equally valid for maxima as well. Refer to the first example in Figure 3. The input points in the example are in two groups: a lower group L, that is not on the upper hull, and an upper group U, arranged so that all points in U are vertices of the upper hull. The sets L and U have the same number of points $n/2$. Suppose that the input distribution $\mathcal{D}$ takes the following form: the points p_1 through $p_{n/2}$ take fixed positions to form U, and for p_i with $i > n/2$, a random point of L is chosen to be p_i (some points of L may be chosen more than once). Thus the "lower" points are essentially a randomly permuted subset of L, with the number of distinct points $\Omega(n)$. The output convex hull will always have vertex set U, while the points of L are all shown to be non-extremal because they are below the line segment with endpoints p_1 and $p_{n/2}$. Thus an optimal algorithm can output the convex hull in $O(n)$ time.

However, in several other algorithmic models, this example needs $\Omega(n \log n)$ time. Since the output size is $n/2$, an output-sensitive algorithm requires $\Omega(n \log n)$. This implies that the *structural entropy* introduced by Barbay [4] is here $\Omega(n \log n)$. Since the expected number of upper hull vertices of a random r-subset of $U \cup L$ is $r/2$, a randomized incremental algorithm takes $\Theta(n \log n)$ time to compute $\mathrm{conv}(I)$ [12]. Because computation of the hull takes linear time for points sorted by a coordinate, say the x-coordinate, it might be thought that self-improving algorithms for sorting [2] would be helpful; however, since the entropy of the sorting output is $\Omega(n \log n)$, such an algorithm would not give a speedup here. By a similar argument, a self-improving algorithm for Delaunay triangulations will also not help. For this input distribution, the entropy of the output Delaunay triangulations is $\Omega(n \log n)$ (because of the randomness among the points of L).

Afshani et al. [1] gave *instance optimal algorithms*, that are "optimal" (in certain restricted models) for *instances* of the planar convex hull problem. In their setup, all inputs from this distribution would be considered "difficult" and would be dealt with in $\Omega(n \log n)$ time. The main difference is that they are interested in an optimal algorithm for any given input considered as a *set*, whereas we want an optimal algorithm for an ensemble of inputs from a special distribution,

where each input is considered as a *sequence*. Indeed, for our algorithm it is essential to know the identity of each point (that is, to know that z_i is the i-th point).

To conclude, we also mention the paradigm of preprocessing imprecise points for faster (Delaunay) triangulation and convex hull computation [6, 16, 19, 22, 24]. Here, we are given a set $\mathcal{R}$ of planar regions, and we would like to preprocess $\mathcal{R}$ in order to quickly find the (Delaunay) triangulation or convex hull for any point set which contains exactly one point from each region in $\mathcal{R}$. This setting is adversarial, but if we only consider point sets where a point is randomly drawn from each region, it can be regarded as a special case of our problem. In this view, these results give us bounds for the eventual running time of a self-improving algorithm if $\mathcal{D}$ draws its points from disjoint planar regions, even though the self-improving algorithm has no prior knowledge of the special input structure. A particularly interesting scenario is the following: given a set of lines L in the plane, we would like to preprocess L such that given any point set P with one point from each line in L, we can compute the convex hull of P quickly. It is possible to achieve near-linear expected time for the convex hull computation, albeit at the expense of a quadratic preprocessing time and storage requirement [16]. Our self-improving algorithm now shows that if each point in P is drawn randomly from a line in L, we can still achieve near-linear expected running time, but with an improved storage requirement of $O(n^{1+\varepsilon})$.

Why is this hard? Since the convex hull is essentially part of the Delaunay triangulation, generally algorithms for the former are simpler than those for the latter; since a self-improving algorithm for Delaunay triangulation was already known, it seems natural to assume that a planar convex hull algorithm in the same model should follow easily, and be simpler than the Delaunay algorithm.[1]

As we explained above, this is not true. Let us try to give some more intuition for this. The reason seems to be the split in output status among the points: some points are simply listed as extremal, and others need a certificate of non-extremality; the certificate may or may not be easy to find. In the first example of Figure 3, the certificates of non-extremality are all "easy". However, if the points of L are placed *just below* the edges of the upper hull, then to find the certificate for a given point in p_i for $i > n/2$, a search must be done to find the single hull edge above p_i, and the certificates are "hard". A simple example shows that even though the points are independent, the convex hull can exhibit very dependent behavior. In the second example of Figure 3, point p_1 can be at either p_h or p_ℓ, but the other points are fixed. The other points become extremal *depending* on the position of p_1. This makes life rather hard for entropy-optimality, since for $p_1 = p_\ell$ the ordering of the remaining points must be determined, but otherwise that is not necessary.

In our algorithm, and plausibly for any algorithm, some form of point location must be done for each p_i. (Here, one search we do involves dividing the plane by x-coordinate, and searching among the resulting vertical slabs.) If point p_i is "easily" shown to be non-extremal, then the point location search should be correspondingly shallow. However, it doesn't seem to be possible to determine, in advance, how deep the point location could go: imagine the points L of Figure 3 doubled up and placed at *both* the "hard" and "easy" positions, and p_i for $i > n/2$ chosen randomly from among those positions; the necessary search depth can only be determined using the position of the point. Also, the certificates may be easy to find, if we know which points are extremal, or at least "near extremal," but determining that is why we are doing the search to begin with.

[1] The authors certainly thought so for awhile, before painfully learning otherwise.

3 Preliminaries and basic facts

First, we give some basic definitions and concepts that will be used throughout the paper.

3.1 Basic notation

We use c to denote a sufficiently large constant. Our input point set is called $P = \langle p_1, \ldots, p_n \rangle$, and it comes from a product distribution $\mathcal{D} = \prod_{i=1}^{n} \mathcal{D}_i$. For any point $p \in P$, we write $x(p)$ and $y(p)$ for the x and y coordinates of p. As mentioned, given a directed line ℓ, we write ℓ^+ for the open halfplane to the left of ℓ, and ℓ^- for the open halfplane to the right of ℓ. If $R \subseteq \mathbb{R}^2$ is a planar region, then a *halving line* for R with respect to a distribution $\mathcal{D}_i$ is a line ℓ such that

$$\Pr_{p \sim \mathcal{D}_i}[p \in \ell^+ \cap R] = \Pr_{p \sim \mathcal{D}_i}[p \in \ell^- \cap R].$$

Note that if $\Pr_{p \sim \mathcal{D}_i}[p \in R] = 0$, every line is a halving line for R.

We use the phrase "with high probability" for any probability larger than $1 - n^{-\Omega(1)}$. The constant buried in the $\Omega(1)$ can be made arbitrarily large by increasing the constant c. We will take union bounds over polynomially many (with a fixed exponent, usually at most n^2) low probability events and still have a low probability bound.

3.2 Linear comparison trees

In this section, we will discuss some basic properties of linear comparison trees. The crucial lemma shows that any linear comparison tree can be converted to a convenient form with only a constant blow-up in depth.

Let $\mathcal{T}$ be a linear comparison tree. We remind the reader that for each node v of $\mathcal{T}$ there exists a set $\mathcal{R}_v \subseteq \mathbb{R}^{2n}$ such that an evaluation of $\mathcal{T}$ on input P reaches v if and only if $P \in \mathcal{R}_v$. The next proposition demonstrates the usefulness of restricted linear comparison trees. They ensure that the sets $\mathcal{R}_v$ are Cartesian products of planar regions, which will later allow us to analyze each input point independently.

Proposition 3.1. *Let $\mathcal{T}$ be a restricted linear comparison tree, and let v be a node in $\mathcal{T}$. Then there exists a sequence $\langle R_1, R_2, \ldots, R_n \rangle$ of (possibly unbounded) convex planar polygons such that $\mathcal{R}_v = \prod_{i=1}^{n} R_i$. In other words, the evaluation of $\mathcal{T}$ on $P = \langle p_1, \ldots, p_n \rangle$ reaches v if and only if $p_i \in R_i$ for all i.*

Proof. The proof is by induction on d_v. For the root, we set $R_1 = \cdots = R_n = \mathbb{R}^2$. If $d_v \geq 1$, let v' be the parent of v. By induction, there are planar convex polygons R'_i with $\mathcal{R}_{v'} = \prod_{i=1}^{n} R'_i$. Furthermore, since $\mathcal{T}$ is restricted, v' is labeled with a test of the form "$p_j \in \ell^+_{v'}$?", where the line $\ell_{v'}$ is independent of the input. Thus, we can take $R_i = R'_i$ for $i \neq j$, and depending on whether v is the left or the right child of v', we set $R_j = R'_j \cap \ell^+_{v'}$ or $R_j = R'_j \cap \ell^-_{v'}$. □

Next, we will show how to restrict the linear comparison trees even further, so that the depth of a node v is related to the probability that v is reached by a random input $P \sim \mathcal{D}$. This will allow us to compare the running times of our algorithms with the depth of a near optimal tree.

Definition 3.2. *Let $\mathcal{T}$ be a restricted linear comparison tree. We call $\mathcal{T}$* entropy-sensitive *if the following holds for every node v of $\mathcal{T}$: let $\mathcal{R}_v = \prod_{i=1}^{n} R_i$ and v be labeled with a test of the form "$p_j \in \ell^+_v$?". Then ℓ_v is a halving line for R_j.*

The following proposition shows that the depth of a node in an entropy-sensitive linear comparison tree is related to the probability that it is being visited.

Proposition 3.3. *Let v be a node in an entropy-sensitive comparison tree, and let $\mathcal{R}_v = \prod_{i_1}^{n} R_i$. Then*

$$d_v = -\sum_{i=1}^{n} \log \Pr_{p_i \sim \mathcal{D}_i}[p_i \in R_i].$$

Proof. We use induction on d_v. The root has depth 0 and all probabilities are 1, so the claim holds in this case. Now let $d_v \geq 1$ and v' be the parent of v. Write $\mathcal{R}_{v'} = \prod_{i=1}^{n} R'_i$ and $\mathcal{R}_v = \prod_{i=1}^{n} R_i$. By induction, we have $d_{v'} = -\sum_{i=1}^{n} \log \Pr[p_i \in R'_i]$. Since $\mathcal{T}$ is entropy-sensitive, v' is labeled with a test of the form "$p_j \in \ell_{v'}^{+}$?", where $\ell_{v'}$ is a halving line for R'_j, i.e., $\Pr[p_j \in R'_j \cap \ell_{v'}^{+}] = \Pr[p_j \in R'_j \cap \ell_{v'}^{-}] = \Pr[p_j \in R'_j]/2$. Since $R_i = R'_i$ for $i \neq j$ and $R_j = R'_j \cap \ell_{v'}^{+}$ or $R_j = R'_j \cap \ell_{v'}^{-}$, it follows that

$$-\sum_{i=1}^{n} \log \Pr_{p_i \sim \mathcal{D}_i}[p_i \in R_i] = 1 - \sum_{i=1}^{n} \log \Pr_{p_i \sim \mathcal{D}_i}[p_i \in R'_i] = 1 + d_{v'} = d_v,$$

as desired. □

In Section 4, we will show that for our purposes, it suffices to restrict our attention to entropy-sensitive comparison trees. The following lemma is a very important part of the proof, as it gives handles on OPT-MAX and OPT-CH.

Lemma 3.4. *Let $\mathcal{T}$ a finite linear comparison tree and $\mathcal{D}$ be a product distribution over points. Then there exists an entropy-sensitive comparison tree $\mathcal{T}'$ with expected depth $d_{\mathcal{D}}(\mathcal{T}') = O(d_{\mathcal{D}}(\mathcal{T}))$, as $d_{\mathcal{D}}(\mathcal{T}) \to \infty$.*

3.3 Search trees and restricted searches

We define the notion of *restricted searches*, central to our proof of optimality. Let $\mathbf{U}$ be an ordered finite set and $\mathcal{F}$ be a distribution over $\mathbf{U}$. For any element $j \in \mathbf{U}$, we write q_j for the probability of j according to $\mathcal{F}$. For any sequence of numbers a_j, $j \in \mathbf{U}$, and for any interval $S \subseteq \mathbf{U}$, we use a_S to denote $\sum_{j \in S} a_j$. Thus, if S is an interval of $\mathbf{U}$, then q_s is the total probability of S.

Let T be a search tree over $\mathbf{U}$. It will be convenient to think of T as (at most) ternary, where each node has at most 2 children that are internal nodes. We associate each internal node v of T with an interval $S_v \subseteq \mathbf{U}$ in the obvious way (any element in S_v encounters v along its search path). In our application of the lemma, $\mathbf{U}$ will just be the set of leaf slabs of a slab structure $\mathbf{S}$, see below. We now introduce some definitions regarding restricted searches and search trees.

Definition 3.5. *Consider an interval S of $\boldsymbol{U}$. An S-*restricted distribution *$\mathcal{F}_S$ is given by the probabilities $\Pr_{\mathcal{F}_S}[j] := s_j / \sum_{r \in \boldsymbol{U}} s_r$ for all $j \in \boldsymbol{U}$, where the s_j, $j \in \boldsymbol{U}$, have the property that $0 \leq s_j \leq q_j$, if $j \in S$; and $s_j = 0$, otherwise.*

*Let $j \in S$. An S-*restricted search *for j is a search for j in T that terminates as soon as it reaches the first node v with $S_v \subseteq S$.*

Definition 3.6. *Let $\mu \in (0, 1)$ be a parameter. A search tree T over $\boldsymbol{U}$ is* μ-reducing *if for any internal node v and for any non-leaf child w of v, we have $q_{S_w} \leq \mu q_{S_v}$.*

A search tree T is α-optimal *for restricted searches over $\mathcal{F}$ if for every interval $S \subseteq \boldsymbol{U}$ and every S-restricted distribution $\mathcal{F}_S$, the expected time of an S-restricted search over $\mathcal{F}_S$ is at most $\alpha(1 - \log s_S)$. (The values s_j are as in Definition 3.5.)*

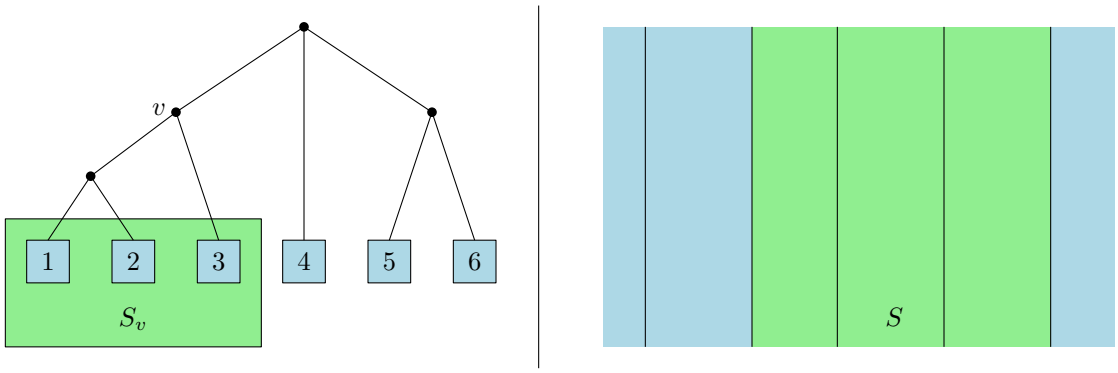

Figure 4: Left: A universe with 6 elements and a search tree on it. The nodes in the tree are ternary, with at most two internal children. The node v corresponds to the interval $S_v = \{1, 2, 3\}$. Right: A vertical slab structure with 6 leaf slabs. (Note that the left and right unbounded slab also count as leaf slabs). S is a (general) slab that consists of 3 leaf slabs, i.e., $|S| = 3$.

We give the main lemma about restricted searches. A tree that is optimal for searches over $\mathcal{F}$ also works for restricted distributions. The proof is given in Section 7.

Lemma 3.7. *Suppose T is a μ-reducing search tree for $\mathcal{F}$. Then T is $O(1/\log(1/\mu))$-optimal for restricted searches over $\mathcal{F}$.*

3.4 Data structures

We describe data structures that will be used in both of our algorithms. A *vertical slab structure* $\mathbf{S}$ is a sequence of vertical lines that partition the plane into vertical regions, called *leaf slabs*. (We will consider the latter to be the open regions between the vertical lines. Since we assume that our distributions are continuous, we abuse notation and consider the leaf slabs to partition the plane.) More generally, a *slab* is the region between any two vertical lines of $\mathbf{S}$. The *size* of a slab S, $|S|$, is the number of leaf slabs it contains. The size of the slab structure is the total number of leaf slabs. We denote it by $|\mathbf{S}|$. Furthermore, for any slab S, the probability that $p_i \sim \mathcal{D}_i$ is in S is denoted by $q(i, S)$.

Both our algorithms will construct special slab structures in the learning phase. Similar structures were first constructed in [2], and we follow the same strategy. All missing proofs are given in Section 8.

Lemma 3.8. *We can construct a slab structure $\mathbf{S}$ with $O(n)$ leaf slabs such that, with probability $1 - n^{-3}$ over the construction of $\mathbf{S}$, the following holds. For a leaf slab λ of $\mathbf{S}$, let X_λ denote the number of points in a random input P that fall into λ. For every leaf slab λ of $\mathbf{S}$, we have $\mathbf{E}[X_\lambda^2] = O(1)$. The construction takes $O(\log n)$ rounds and $O(n \log^2 n)$ time.*

The algorithms construct a collection of specialized search trees for $\mathbf{S}$ for each distribution $\mathcal{D}_i$. It is important that these trees can be represented in small space. The following lemma gives the details of the constructed search trees.

Lemma 3.9. *Let $\varepsilon > 0$ be a fixed parameter and $\boldsymbol{S}$ a slab structure with $O(n)$ leaf slabs. In $O(n^\varepsilon)$ rounds and $O(n^{1+\varepsilon})$ time, we can construct search trees $T_1, T_2, \ldots, T_n$ over $\boldsymbol{S}$ such that the*

following holds: (i) the trees can be represented in $O(n^{1+\varepsilon})$ total space; (ii) with probability $1-n^{-3}$ over the construction of the T_is, every T_i is $O(1/\varepsilon)$-optimal for restricted searches over $\mathcal{D}_i$.

We will also require a simple structure that maintains (key, index) pairs. The indices are all distinct and always in $[n]$. The keys are elements from the ordered universe $[|\mathbf{S}|]$. We store these pairs in a data structure that allows the operations `insert`, `delete` (deleting a pair), `find-max` (finding the maximum key among the stored pairs), and `decrease-key` (decreasing the key of some stored pair). For `decrease` and `decrease-key`, we assume the input is a pointer into the data structure to the stored pair to operate on.

Claim 3.10. *Suppose there are x `find-max` operations and y `decrease-key` operations. We can implement the data structure $L(A)$ such that the total time for the operations is $O(n+x+y)$. The storage requirement is $O(n)$.*

Proof. We represent $L(A)$ as an array of lists. For every $k \in [|\mathbf{S}|]$, we keep a list of indices whose key values are k. We maintain m, the current maximum key. The total storage is $O(n)$. A `find-max` trivially takes $O(1)$ time, and an `insert` is done by adding the element to the appropriate list. For a `delete`, we remove the element from the list (assuming appropriate pointers are available). We now have to update the maximum. If the list at m is non-empty, no action is required. If it is empty, we check sequentially whether the list at $m-1, m-2, \ldots$ is empty. This will eventually lead to the maximum. To do a `decrease-key`, we `delete`, `insert`, and then update the maximum.

Note that since all key updates are `decrease-key`s, the maximum can only decrease. Hence, the total overhead for scanning for a new maximum is $O(n)$. □

4 Linear comparison trees

A major challenge of self-improving algorithms is the strong requirement of optimality for the distribution $\mathcal{D}$. We focus on the model of linear comparison trees, and let $\mathcal{T}$ be an optimal tree for distribution $\mathcal{D}$. (There may be distributions where such an exact $\mathcal{T}$ does not exist, but we can always find one that is nearly optimal.) One of our key insights is that when $\mathcal{D}$ is a product distribution, we can convert $\mathcal{T}$ to a restricted comparison tree whose expected depth is only a constant factor worse. In other words, there always exists a near-optimal restricted comparison tree for our problem. Furthermore, we will see that this tree can be made entropy-sensitive.

In such a tree, each leaf v is labeled with a sequence of regions $\mathcal{R}_v = (R_1, R_2, \ldots, R_n)$. Any input $P = (p_1, p_2, \ldots, p_n)$ such that $p_i \in R_i$ for all i, will lead to v. Since the distributions are independent, we can argue that the probability that an input leads to v is $\prod_i \Pr_{p_i \sim \mathcal{D}_i}[p_i \in R_i]$. Furthermore, by entropy-sensitivity, the depth of v is $-\sum_i \log \Pr[p_i \in R_i]$. This gives us a concrete bound that we can exploit.

It now remains to show that if we start with a random input from $\mathcal{R}_v$, the expected running time is bounded by the sum given above. We will argue that for such an input, as soon as the search for p_i locates it inside R_i, the search will terminate. This leads to the optimal running time.

The purpose of this section is to prove Lemma 3.4. This is done in two steps. First, we go from a linear comparison tree to a restricted linear comparison tree.

Lemma 4.1. *Let $\mathcal{T}$ a finite linear comparison tree and $\mathcal{D}$ be a product distribution over points. Then there exists a restricted comparison tree $\mathcal{T}'$ with expected depth $d_{\mathcal{D}}(\mathcal{T}') = O(d_{\mathcal{D}}(\mathcal{T}))$, as $d_{\mathcal{D}}(\mathcal{T}) \to \infty$.*

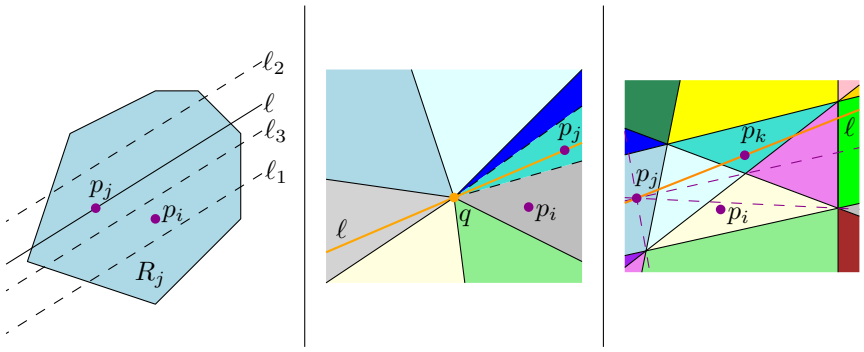

Figure 5: The different cases in the proof of Claim 4.3.

The proof of Lemma 4.1 is in Section 4.1. Next, we show how to go from a restricted linear comparison tree to an entropy-sensitive comparison tree.

Lemma 4.2. *Let $\mathcal{T}$ a restricted linear comparison tree. Then there exists an entropy-sensitive comparison tree $\mathcal{T}'$ with expected depth $d_{\mathcal{T}'} = O(d_{\mathcal{T}})$.*

The proof of Lemma 4.1 is in Section 4.2. Given Lemmas 4.1 and 4.2, the proof Lemma 3.4 is immediate by applying the lemmas in succession.

4.1 Reducing to restricted comparison trees

We will describe a transformation from $\mathcal{T}$ into a restricted comparison tree with similar depth. The first step is to show how to represent a single comparison by a restricted linear comparison tree, provided that P is drawn from a product distribution. The final transformation basically replaces each node of $\mathcal{T}$ by the subtree given by the next claim. For convenience, we will drop the subscript of $\mathcal{D}$ from $d_{\mathcal{D}}$, since we focus on a fixed distribution.

Claim 4.3. *Consider a comparison C as described in Definition 2.3, where the comparisons are listed in increasing order of complexity. Let $\mathcal{D}'$ be a product distribution for P such that each p_i is drawn from a polygonal region R_i. Then either C is the simplest, type (1) comparison, or there exists a restricted linear comparison tree $\mathcal{T}'_C$ that resolves the comparison C such that the expected depth of $\mathcal{T}'_C$ (over the distribution $\mathcal{D}'$) is $O(1)$, and all comparisons used in $\mathcal{T}'_C$ are less complex than C.*

Proof. v **is of type (2).** This means that v needs to determine whether an input point p_i lies to the left of the directed line ℓ through another input point p_j with a fixed slope a. We replace this comparison with a binary search. Let R_j be the region in $\mathcal{D}'$ corresponding to p_j. Take a halving line ℓ_1 for R_j with slope a. Then perform two comparisons to determine on which side of ℓ_1 the inputs p_i and p_j lie. If p_i and p_j lie on different sides of ℓ_1, we declare success and resolve the original comparison accordingly. Otherwise, we replace R_j with the appropriate new region and repeat the process until we can declare success. Note that in each attempt the success probability is at least $1/4$. The resulting restricted tree $\mathcal{T}'_C$ can be infinite. Nonetheless, the probability that an evaluation of $\mathcal{T}'_C$ leads to a node of depth k is at most $2^{-\Omega(k)}$, so the expected depth is $O(1)$.

v is of type (3). Here the node v needs to determine whether an input point p_i lies to the left of the directed line ℓ through another input point p_j and a fixed point q.

We partition the plane by a constant-sized family of cones, each with apex q, such that for each cone V in the family, the probability that line $\overline{qp_j}$ meets V (other than at q) is at most $1/2$. Such a family could be constructed by a sweeping a line around q, or by taking a sufficiently large, but constant-sized, sample from the distribution of p_j, and bounding the cones by all lines through q and each point of the sample. Such a construction has a non-zero probability of success, and therefore the described family of cones exists.

We build a restricted tree that locates a point in the corresponding cone. For each cone V, we can recursively build such a family of cones (inside V), and build a tree for this structure as well. Repeating for each cone, this leads to an infinite restricted tree $\mathcal{T}'_C$. We search for both p_i and p_j in $\mathcal{T}'_C$. When we locate p_i and p_j in two different cones of the same family, then comparison between p_i and $\overline{qp_j}$ is resolved and the search terminates. The probability that they lie in the same cones of a given family is at most $1/2$, so the probability that the evaluation leads to k steps is at most $2^{-\Omega(k)}$.

v is of type (4). Here the node v needs to determine whether an input point p_i lies to the left of the directed line ℓ through input points p_j and p_k.

We partition the plane by a constant-sized family of triangles and cones, such that for each region V in the family, the probability that the line through p_j and p_k meets V is at most $1/2$. Such a family could be constructed by taking a sufficiently large random sample of pairs p_j and p_k and triangulating the arrangement of the lines through each pair. Such a construction has a non-zero probability of success, and therefore such a family exists. (Other than the source of the random lines used in the construction, this scheme goes back at least to [10]; a tighter version, called a *cutting*, could also be used [8].)

When computing C, suppose p_i is in region V of the family. If the line $\overline{p_jp_k}$ does not meet V, then the comparison outcome is known immediately. This occurs with probability at least $1/2$. Moreover, determining the region containing p_i can be done with a constant number of comparisons of type (1), and determining if $\overline{p_jp_k}$ meets V can be done with a constant number of comparisons of type (3); for the latter, suppose V is a triangle. If $p_j \in V$, then $\overline{p_jp_k}$ meets V. Otherwise, suppose p_k is above all the lines through p_j and each vertex of V; then $\overline{p_jp_k}$ does not meet V. Also, if p_k is below all the lines through p_j and each vertex, then $\overline{p_jp_k}$ does not meet V. Otherwise, $\overline{p_jp_k}$ meets V. So a constant number of type (1) and type (3) queries suffice.

By recursively building a tree for each region V of the family, comparisons of type (4) can be done via a tree whose nodes use comparisons of type (1) and (3) only. Since the probability of resolving the comparison is at least $1/2$ with each family of regions that is visited, the expected number of nodes visited is constant. □

Given Claim 4.3, it is now easy to prove Lemma 4.1.

Proof of Lemma 4.1. We transform $\mathcal{T}$ into a tree $\mathcal{T}'$ that has no comparisons of type (4), by using the construction of Claim 4.3 where nodes of type (4) are replaced by a tree. We then transform $\mathcal{T}'$ into a tree $\mathcal{T}''$ that has no comparisons of type (3) or (4), and finally transform $\mathcal{T}'''$ into a restricted tree. Each such transformation is done in the same general way, using one case of Claim 4.3, so we focus on the first one.

We incrementally transform $\mathcal{T}$ into the tree $\mathcal{T}'$. In each step, we have a partial restricted comparison tree $\mathcal{T}''$ that will eventually become $\mathcal{T}'$. Furthermore, during the process each node of

$\mathcal{T}$ is in one of three different states. It is either *finished*, *fringe*, or *untouched*. Finally, we have a function S that assigns to each finished and to each fringe node of $\mathcal{T}$ a subset $S(v)$ of nodes in $\mathcal{T}''$.

The initial situation is as follows: all nodes of $\mathcal{T}$ are untouched except for the root which is fringe. Furthermore, the partial tree $\mathcal{T}''$ consists of a single root node r and the function S assigns the root of $\mathcal{T}$ to the set $\{r\}$.

Now our transformation proceeds as follows. We pick a fringe node v in $\mathcal{T}$, and mark v as finished. For each child v' of v, if v' is an internal node of $\mathcal{T}$, we mark it as fringe. Otherwise, we mark v' as finished. Next, we apply Claim 4.3 to each node $w \in S(v)$. Note that this is a valid application of the claim, since w is a node of $\mathcal{T}''$, a restricted tree. Hence $\mathcal{R}_w$ is a product set, and the distribution $\mathcal{D}$ restricted to $\mathcal{R}_w$ is a product distribution. Hence, replace each node $w \in S(v)$ in $\mathcal{T}''$ by the subtree given by Claim 4.3. Now $S(v)$ contains the roots of these subtrees. Each leaf of each such subtree corresponds to an outcome of the comparison in v. (Potentially, the subtrees are countably infinite, but the expected number of steps to reach a leaf is constant.) For each child v' of v, we define $S(v')$ as the set of all such leaves that correspond to the same outcome of the comparison as v'. We continue this process until there are no fringe nodes left. By construction, the resulting tree $\mathcal{T}'$ is restricted.

It remains to argue that $d_{\mathcal{T}'} = O(d_{\mathcal{T}})$.

Let v be a node of $\mathcal{T}$. We define two random variables X_v and Y_v. The variable X_v is the indicator random variable for the event that the node v is traversed for a random input $P \sim \mathcal{D}$. The variable Y_v denotes the number of nodes traversed in $\mathcal{T}'$ that correspond to v (i.e., the number of nodes needed to simulate the comparison at v, if it occurs). We have $d_{\mathcal{T}} = \sum_{v \in \mathcal{T}} \mathbf{E}[X_v]$, because if the leaf corresponding to an input $P \sim \mathcal{D}$ has depth d, exactly d nodes are traversed to reach it. We also have $d_{\mathcal{T}'} = \sum_{v \in \mathcal{T}} \mathbf{E}[Y_v]$, since each node in $\mathcal{T}'$ corresponds to exactly one node v in $\mathcal{T}$. Claim 4.4 below shows that $\mathbf{E}[Y_v] = O(\mathbf{E}[X_v])$, which completes the proof. □

Claim 4.4. $\mathbf{E}[Y_v] \leq c\mathbf{E}[X_v]$

Proof. Note that $\mathbf{E}[X_v] = \Pr[X_v = 1] = \Pr[P \in \mathcal{R}_v]$. Since the sets $\mathcal{R}_w$, $w \in S(v)$, partition $\mathcal{R}_v$, we can write $\mathbf{E}[Y_v]$ as

$$\mathbf{E}[Y_v \mid X_v = 0] \Pr[X_v = 0] + \sum_{w \in S(v)} \mathbf{E}[Y_v \mid P \in \mathcal{R}_w] \Pr[P \in \mathcal{R}_w].$$

Since $Y_v = 0$ if $P \notin \mathcal{R}_v$, we have $\mathbf{E}[Y_v \mid X_v = 0] = 0$. Also, $\Pr[P \in \mathcal{R}_v] = \sum_{w \in S(v)} \Pr[P \in \mathcal{R}_w]$. Furthermore, by Claim 4.3, we have $\mathbf{E}[Y_v \mid P \in \mathcal{R}_w] \leq c$. The claim follows. □

4.2 Entropy-sensitive comparison trees

We now prove Lemma 4.2. The proof extends the proof of Lemma 4.1, via an extension to Claim 4.3. We can regard a comparison against a fixed halving line as simpler than an comparison against an arbitrary fixed line. Our extension of Claim 4.3 is the claim that any type (1) node can be replaced by a tree with constant expected depth, as follows. A comparison $p_i \in \ell^+$ can be replaced by a sequence of comparisons to halving lines. Similar to the reduction for type (2) comparisons in Claim 4.3, this is done by binary search. That is, let ℓ_1 be a halving line for R_i parallel to ℓ. We compare p_i with ℓ. If this resolves the original comparison, we declare success. Otherwise, we repeat the process with the halving line for the new region R_i'. In each step, the probability of success is at least $1/2$. The resulting comparison tree has constant expected depth; we now apply

the construction of Lemma 4.1 to argue that for a restricted tree $\mathcal{T}$ there is an entropy-sensitive version $\mathcal{T}'$ whose expected depth is larger by at most a constant factor.

5 A self-improving algorithm for coordinate-wise maxima

We begin with an informal overview of the algorithm.

If the points of P are sorted by x-coordinate, the maxima of P can be found easily by a right-to-left sweep over P: we maintain the largest y-coordinate Y of the points traversed so far; when a point p is visited in the traversal, if $y(p) < Y$, then p is non-maximal, and the point p_j with $Y = y(p_j)$ gives a per-point certificate for p's non-maximality. If $y(p) \geq Y$, then p is maximal, and we can update Y and put p at the beginning of the certificate list of maxima of P.

This suggests the following approach to a self-improving algorithm for maxima: sort P with a self-improving sorter and then use the traversal. The self-improving sorter of [2] works by locating each point of P within the slab structure $\mathbf{S}$ of Lemma 3.8 using the trees T_i of Lemma 3.9.

While this approach does use $\mathbf{S}$ and the T_i's, it is not optimal for maxima, because the time spent finding the exact sorted order of non-maximal points may be wasted: in some sense, we are learning much more information about the input P than necessary. To deduce the list of maxima, we do not need the sorted order of *all* points of P: it suffices to know the sorted order of just the maxima! An optimal algorithm would probably locate the maximal points in $\mathbf{S}$ and would not bother locating "extremely non-maximal" points. This is, in some sense, the difficulty that output-sensitive algorithms face.

As a thought experiment, let us suppose that the maximal points of P are known to us, but not in sorted order. We search only for these in $\mathbf{S}$ and determine the sorted list of maximal points. We can argue that the optimal algorithm must also (in essence) perform such a search. We also need to find per-point certificates for the non-maximal points. We use the slab structure $\mathbf{S}$ and the search trees, but now we shall be very conservative in our searches. Consider the search for a point p_i. At any intermediate stage of the search, p_i is placed in a slab S. This rough knowledge of p_i's location may already suffice to certify its non-maximality: let m denote the leftmost maximal point to the right of S (since the sorted list of maxima is known, this information can be easily deduced). We check if m dominates p_i. If so, we have a per-point certificate for p_i and we promptly terminate the search for p_i. Otherwise, we continue the search by a single step and repeat. We expect that many searches will not proceed too long, achieving a better position to compete with the optimal algorithm.

Non-maximal points that are dominated by many maximal points will usually have a very short search. Points that are "nearly" maximal will require a much longer search. So this approach should derive just the "right" amount of information to determine the maxima output. But wait! Didn't we assume that the maximal points were known? Wasn't this crucial in cutting down the search time? This is too much of an assumption, and because the maxima are highly dependent on each other, it is not clear how to determine which points are maximal before performing searches.

The final algorithm overcomes this difficulty by interleaving the searches for sorting the points with confirmation of the maximality of some points, in a rough right-to-left order that is a more elaborate version of the traversal scheme given above for sorted points. The searches for all points p_i (in their respective trees T_i) are performed "together", and their order is carefully chosen. At any intermediate stage, each point p_i is located in some slab S_i, represented by some node of its search tree. We choose a specific point and advance its search by one step. This order is very

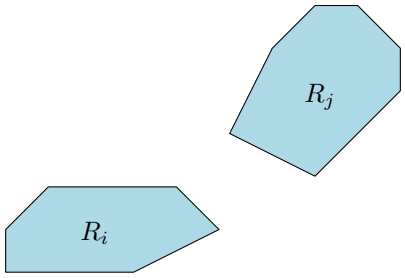

Figure 6: Every point in R_j dominates every point in R_i.

important, and is the basis of our optimality. The algorithm is described in detail and analyzed in Section 5.2.

5.1 Optimality

We use a more restricted form of certificates and search trees that allows for easier proofs of optimality.

Proposition 5.1. *Consider a leaf v of a restricted linear comparison tree $\mathcal{T}$ computing the maxima. Let R_i be the region associated with non-maximal point $p_i \in P$ in $\mathcal{R}_v$. There exists some region R_j associated with an extremal point p_j such that every point in R_j dominates every point in R_i.*

Proof. The leaf v is associated with a certificate γ, such that γ is valid for every input that reaches v. Consider the point $p_i \in P$. The certificate γ associates the non-maximal point p_i with p_j such that p_j dominates p_i. For any input P reaching v, p_j dominates p_i. Let us first argue that p_j can be assumed to be maximal. Construct a digraph G over the vertex set $[n]$ where directed edge (u, v) is present if (according to γ), p_u is dominated by p_v. All vertices have an outdegree of 1, and there are no cycles in G (since domination is a transitive relationship). Hence, G is a forest of trees with edges directed towards the root. The roots are all maximal vertices, and any point in a subtree is dominated by the point corresponding to the root. Therefore, we can rewrite γ so that all dominating points are extremal.

Since $\mathcal{T}$ is restricted, the region $\mathcal{R}_v \subseteq \mathbb{R}^{2n}$ corresponding to v is a Cartesian product of polygonal regions $(R_1, R_2, \ldots, R_n)$. Suppose there exist two subregions $R'_i \subseteq R_i$ and $R'_j \subseteq R_j$ such that R'_j does not dominate R'_i. Consider the input P where $p_i \in R'_i$ and $p_j \in R'_j$. The remaining points are arbitrarily chosen in respective R_k's. The certificate γ is not valid for P, contradicting the nature of $\mathcal{T}$. Hence, all points in R_j dominate all points in R_i, and p_j is extremal. □

We now enhance the notion of a certificate (Definition 2.1) to make it more useful for our algorithm's analysis. For technical reasons, we want points to be "well-separated" according to the slab structure $\mathbf{S}$. By Proposition 5.1, every non-maximal point is associated with a dominating region.

Definition 5.2. *Let $\boldsymbol{S}$ be a slab structure. A certificate for an input P is called* $\mathbf{S}$-labeled *if the following holds. Every maximal point is labeled with the leaf slab of $\boldsymbol{S}$ containing it. Every non-maximal point is either placed in the containing leaf slab, or is separated from a dominating region by a slab boundary.*

We say that a tree $\mathcal{T}$ *computes the* $\mathbf{S}$*-labeled maxima* if the leaves are labeled with $\mathbf{S}$-labeled certificates.

Lemma 5.3. *There exists an entropy-sensitive comparison tree $\mathcal{T}$ computing the $\mathbf{S}$-labeled maxima whose expected depth over $\mathcal{D}$ is $O(n + \text{OPT-MAX}_{\mathcal{D}})$.*

Proof. Start with an optimal linear comparison tree $\mathcal{T}'$ that computes the maxima. At every leaf, we have a list M with the maximal points in sorted order. We merge M with the list of slab boundaries of $\mathbf{S}$ to label each maximal point with the leaf slab of $\mathbf{S}$ containing it. We now deal with the non-maximal points. Let R_i be the region associated with a non-maximal point p_i, and R_j be the dominating region. Let λ be the leaf slab containing R_j. Note that the x-projection of R_i cannot extend to the right of λ. If there is no slab boundary separating R_i from R_j, then R_i must intersect λ. With one more comparison, we can place p_i inside λ or strictly to the left of it. All in all, with $O(n)$ more comparisons than $\mathcal{T}'$, we have a tree $\mathcal{T}''$ that computes the $\mathbf{S}$-labeled maxima. Hence, the expected depth is $\text{OPT-MAX}_{\mathcal{D}} + O(n)$. Now we apply Lemma 3.4 to $\mathcal{T}''$ to get an entropy-sensitive comparison tree $\mathcal{T}$ computing the $\mathbf{S}$-labeled maxima with expected depth $O(n + \text{OPT-MAX}_{\mathcal{D}})$. □

5.2 The algorithm

In the learning phase, the algorithm constructs a slab structure $\mathbf{S}$ and search trees T_i, as given in Lemmas 3.8 and 3.9. Henceforth, we assume that we have these data structures, and will describe the algorithm in the limiting (or stationary) phase. Our algorithm proceeds by searching progressively each point p_i in its tree T_i. However, we need to choose the order of the searches carefully.

At any stage of the algorithm, each point p_i is placed in some slab S_i. The algorithm maintains a set A of *active points*. An inactive point is either proven to be non-maximal, or it has been placed in a leaf slab. The active points are stored in a data structure $L(A)$, as in Claim 3.10. Recall that $L(A)$ supports the operations `insert`, `delete`, `decrease-key`, and `find-max`. The key associated with an active point p_i is the right boundary of the slab S_i (represented as an element of $[|\mathbf{S}|]$).

We list the variables that the algorithm maintains. The algorithm is initialized with $A = P$, and each S_i is the largest slab in $\mathbf{S}$. Hence, all points have key $|\mathbf{S}|$, and we `insert` all these keys into $L(A)$.

1. $A, L(A)$: the list A of active points stored in data structure $L(A)$.
2. $\widehat{\lambda}, B$: Let m be the largest key among the active points. Then $\widehat{\lambda}$ is the leaf slab whose right boundary is m and B is a set of points located in $\widehat{\lambda}$. Initially B is empty and m is $|S|$, corresponding to the $+\infty$ boundary of the rightmost, infinite, slab.
3. $M, \hat{p}$: M is a sorted (partial) list of currently discovered maximal points and $\hat{p}$ is the leftmost among those. Initially M is empty and $\hat{p}$ is a "null" point that dominates no input point.

The algorithm involves a main procedure **Search**, and an auxiliary procedure **Update**. The procedure **Search** chooses a point and proceeds its search by a single step in the appropriate tree. Occasionally, it will invoke **Update** to change the global variables. The algorithm repeatedly calls

Search until $L(A)$ is empty. After that, we perform a final call to **Update** in order to process any points that might still remain in B.

Search. Let p_i be obtained by performing a `find-max` in $L(A)$. If the maximum key m in $L(A)$ is less than the right boundary of $\widehat{\lambda}$, we invoke **Update**. If p_i is dominated by $\hat{p}$, we delete p_i from $L(A)$. If not, we advance the search of p_i in T_i by a single step, if possible. This updates the slab S_i. If the right boundary of S_i has decreased, we perform the appropriate `decrease-key` operation on $L(A)$. (Otherwise, we do nothing.)

Suppose the point p_i reaches a leaf slab λ. If $\lambda = \widehat{\lambda}$, we remove p_i from $L(A)$ and insert it in B (in time $O(|B|)$). Otherwise, we leave p_i in $L(A)$.

Update. We sort all the points in B and update the list of current maxima. As Claim 5.4 will show, we have the sorted list of maxima to the right of $\widehat{\lambda}$. Hence, we can append to this list in $O(|B|)$ time. We reset $B = \emptyset$, set $\widehat{\lambda}$ to the leaf slab to the left of m, and return.

The following claim states an important invariant maintained by the algorithm, and then give a construction for the data structure $L(A)$.

Claim 5.4. *At any time in the algorithm, the maxima of all points to the right of $\widehat{\lambda}$ have been determined in sorted order.*

Proof. The proof is by backward induction on m, the right boundary of $\widehat{\lambda}$. When $m = |S|$, then this is trivially true. Let us assume it is true for a given value of m, and trace the algorithm's behavior until the maximum key becomes smaller than m (which is done in **Update**). When **Search** processes a point p with a key of m then either (i) the key value decreases; (ii) p is dominated by $\hat{p}$; or (iii) p is eventually placed in $\widehat{\lambda}$ (whose right boundary is m). In all cases, when the maximum key decreases below m, all points in $\widehat{\lambda}$ are either proven to be non-maximal or are in B. By the induction hypothesis, we already have a sorted list of maxima to the right of m. The procedure **Update** will sort the points in B and all maximal points to the right of $m-1$ will be determined. □

5.2.1 Running time analysis

The aim of this section is to prove the following lemma.

Lemma 5.5. *The algorithm runs in $O(n + \text{OPT-MAX}_{\mathcal{D}})$ time.*

We can easily bound the running time of all calls to **Update**.

Claim 5.6. *The expected time for all calls to* **Update** *is $O(n)$.*

Proof. The total time for all calls to **Update** is at most the time for sorting points within the leaf slabs. By Lemma 3.8, this takes expected time

$$\mathbf{E}\Big[\sum_{\lambda \in \mathbf{S}} X_\lambda^2\Big] = \sum_{\lambda \in \mathbf{S}} \mathbf{E}\big[X_\lambda^2\big] = \sum_{\lambda \in \mathbf{S}} O(1) = O(n).$$

□

The important claim is the following, since it allows us to relate the time spent by **Search** to the entropy-sensitive comparison trees. Lemma 5.5 follows directly from this.

Claim 5.7. *Let $\mathcal{T}$ be an entropy-sensitive comparison tree computing $\boldsymbol{S}$-labeled maxima. Consider a leaf v labeled with the regions $\mathcal{R}_v = (R_1, R_2, \ldots, R_n)$, and let d_v denote the depth of v. Conditioned on $P \in \mathcal{R}_v$, the expected running time of* **Search** *is $O(n + d_v)$.*

Proof. For each R_i, let S_i be the smallest slab of $\mathbf{S}$ that completely contains R_i. We will show that the algorithm performs at most an S_i-restricted search for input $P \in \mathcal{R}_v$. If p_i is maximal, then R_i is contained in a leaf slab (this is because the output is $\mathbf{S}$-labeled). Hence S_i is a leaf slab and an S_i-restricted search for a maximal p_i is just a complete search.

Now consider a non-maximal p_i. By the properties of $\mathbf{S}$-labeled maxima, the associated region R_i is either inside a leaf slab or is separated by a slab boundary from the dominating region R_j. In the former case, an S_i-restricted search is a complete search. In the latter case, we argue that an S_i-restricted search suffices to process p_i. This follows from Claim 5.4: by the time an S_i-restricted search finishes, all maxima to the right of S_i have been determined. In particular, we have found p_j, and thus $\hat{p}$ dominates p_i. Hence, the search for p_i will proceed no further.

The expected search time taken conditioned on $P \in \mathcal{R}_v$ is the sum (over i) of the conditional expected S_i-restricted search times. Let $\mathcal{E}_i$ denote the event that $p_i \in R_i$, and $\mathcal{E}$ be the event that $P \in \mathcal{R}_v$. We have $\mathcal{E} = \bigwedge_i \mathcal{E}_i$. By the independence of the distributions and linearity of expectation

$$\begin{aligned}\mathbf{E}_{\mathcal{E}}[\text{search time}] &= \sum_{i=1}^{n} \mathbf{E}_{\mathcal{E}}[S_i\text{-restricted search time for } p_i] \\ &= \sum_{i=1}^{n} \mathbf{E}_{\mathcal{E}_i}[S_i\text{-restricted search time for } p_i].\end{aligned}$$

By Lemma 3.7, the time for an S_i-restricted search conditioned on $p_i \in R_i$ is $O(-\log \Pr[p_i \in R_i]+1)$. By Proposition 3.3, $d_v = \sum_i -\log \Pr[p_i \in R_i]$, completing the proof. □

We can now prove the main lemma.

Proof of Lemma 5.5. By Lemma 5.3, there exists an entropy-sensitive comparison tree $\mathcal{T}$ that computes the $\mathbf{S}$-labeled maxima with expected depth $O(\text{OPT-MAX} + n)$. According to Claim 5.7, the expected running time of **Search** is $O(\text{OPT-MAX} + n)$. Claim 5.6 tells us the expected time for **Update** is $O(n)$, and we add these bounds to complete the proof. □

6 A self-improving algorithm for convex hulls

We begin with an outline of the main ideas. The basic philosophy is the same as with the interleaved search for maxima. We set up a slab structure $\mathbf{S}$, and each distribution has a dedicated tree for searching points. At any stage, each point is in some intermediate node of the search tree, and we wish to proceed searches for points that have the greatest potential for being on the convex hull. Furthermore, we would like to quickly ascertain that a point is not extremal, so that we can terminate its search.

For maxima, this strategy is easy enough to implement. The "rightmost" point (among those being searched) is a good candidate for being maximal and we always proceed its search. We also maintain the leftmost maximal point currently known. Any time we continue the search of some point, we can always compare with this maximal point. Hence, any point that is dominated by the current set of maximal points is immediately removed.

For convex hulls, this is much more problematic. At any stage, there are many points with the potential of being extremal, and it is not clear how to choose between them. We also need a procedure that can be rapidly updated as we find new extremal points, so that we quickly certify non-extremal points that are currently being searched.

To perform these operations, we construct a *canonical hull* $\mathcal{C}$ in the learning phase. Roughly speaking, this is a very crude guide for some properties of the actual convex hull. The canonical hull has two key properties. First, any point that is below $\mathcal{C}$ is likely to be non-extremal. Second, there are not too many points above $\mathcal{C}$. We build the slab structure $\mathbf{S}$ based on $\mathcal{C}$. The searches only place points above or below $\mathcal{C}$. A point p is proven to be below $\mathcal{C}$ if we find a segment contained in $\mathcal{C}$ above p. For all points p above $\mathcal{C}$, we perform more searching to find all edges of $\mathcal{C}$ visible from p (think of this as completely certifying that p is above $\mathcal{C}$). This procedure is referred to as the *location algorithm*. At the end of this, we have some partial information about the various points. Then we apply a *construction algorithm*, that computes the convex hull using this information.

6.1 The canonical directions

This section describes all the structures obtained in the learning phase. In order to characterize the typical behavior of a random point set $P \sim \mathcal{D}$, we use a set $\mathbf{V}$ of *canonical directions*. A *direction* is a two-dimensional unit vector, and directions are ordered clockwise. The directions we consider will always point upwards. Given a direction v, we say that $p \in P$ is *extremal* for v if the scalar product $\langle p, v\rangle$ is maximum among all points in P. We denote the lexicographically smallest input point that is extremal for v by e_v. The canonical directions are characterized by the following lemma, whose proof is postponed to Section 9.1. They are computed in the learning phase. (Some of the basic notation used here is given by Definition 2.2 and just above it.)

Lemma 6.1. *Let $k := n/\log^2 n$. There is an $O(n \operatorname{poly}(\log n))$ procedure that requires $\operatorname{poly}(\log n)$ random inputs and outputs an ordered sequence $\mathbf{V} = v_1, v_2, \ldots, v_k$ of directions such that the following holds (with probability at least $1 - n^{-4}$ over construction). Let $P \sim \mathcal{D}$ be a random input. For $i = 1, \ldots, k$, let $e_i := e_{v_i} \in P$, let X_i be the number of points from P inside $uss(e_i, e_{i+1})$, and Y_i the number of* extremal *points inside $uss(e_i, e_{i+1})$. Then*

$$\mathbf{E}_{P\sim\mathcal{D}}\Big[\sum_{i=1}^{k} X_i \log(Y_i + 1)\Big] = O(n \log\log n).$$

Given the canonical directions, we construct some special lines normal to them in the learning phase. The details are given in Section 9.2.

Lemma 6.2. *We can construct (in $O(n \operatorname{poly}(\log n))$ time with a single sample input) lines ℓ_1, ℓ_2, ..., ℓ_k, with ℓ_i normal to e_i, and with the following property (with probability at least $1 - n^{-4}$ over construction). For $i = 1, \ldots, k$, (and sufficiently large constant c)*

$$\Pr_{P\sim\mathcal{D}}[|\ell_i^+ \cap P| \in [1, c \log n]] \geq 1 - n^{-3}.$$

We will henceforth assume that the learning phase succeeds, so the directions and lines obtained have the desired properties. We say that $p \in P$ is $\mathbf{V}$*-extremal* if $p = e_v$ for some $v \in \mathbf{V}$. Using the canonical directions from Lemma 6.1 and the lines from Lemma 6.2, we construct a *canonical hull* $\mathcal{C}$ that is meant to be "typical" for a random $P \sim \mathcal{D}$. We define the canonical hull $\mathcal{C}$ as the intersection

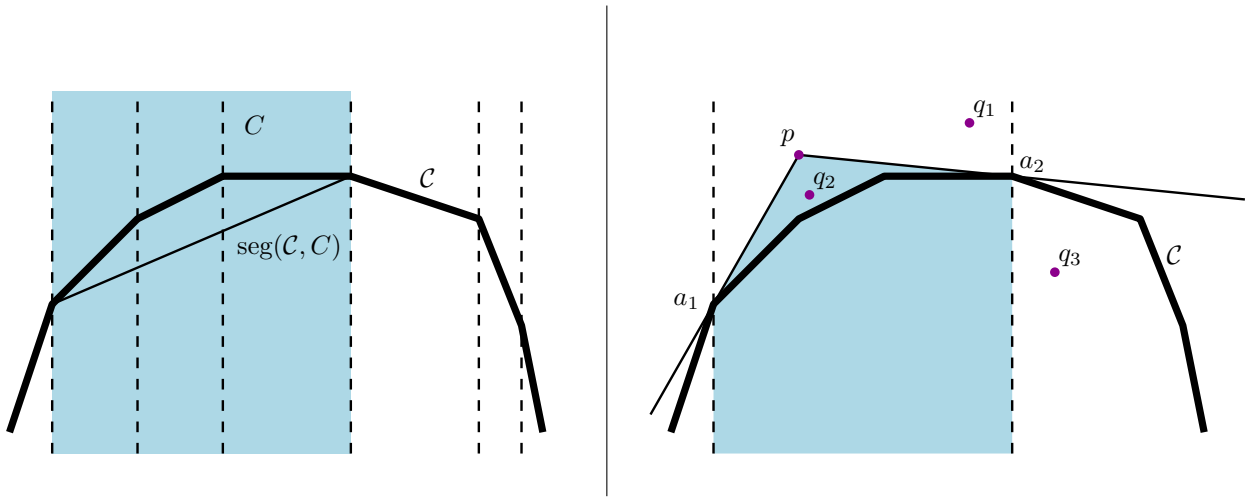

Figure 7: Left: The $\mathcal{C}$-leaf slabs are shown in dashed lines. The shaded portion corresponds to a $\mathcal{C}$-slab C. Right: The shaded region is the pencil of p and the area between the dashed lines in the pencil slab. The point q_1 is above the pencil, q_2 lies inside it, and q_3 is not comparable to it.

of the halfplanes below the ℓ_i, i.e., $\mathcal{C} := \bigcap_{i=1}^{k} \ell_i^-$. It is a convex polygonal region bounded by the lines ℓ_i. We state a simple corollary, that holds by taking a union bound of Lemma 6.2 over all i. Observe that it also implies that the total number of points outside $\mathcal{C}$ is $O(n/\log n)$.

Corollary 6.3. *Assume the learning phase succeeds. With probability at least $1-n^{-2}$, the following holds. For all i, the extremal point for v_i is outside $\mathcal{C}$. The number of pairs (p, s), where $p \in P \setminus \mathcal{C}$, s is an edge of $\mathcal{C}$, and s is visible from p is $O(n/\log n)$.*

We now list some preliminary concepts related to $\mathcal{C}$, see Figure 7. By drawing a vertical line through each vertex of $\mathcal{C}$, we obtain a subdivision of the plane into vertical slabs. We call these slabs the *$\mathcal{C}$-leaf-slabs*. A contiguous interval of $\mathcal{C}$-leaf slabs again forms a vertical slab, which we call a *$\mathcal{C}$-slab*. The $\mathcal{C}$-leaf-slabs constitute the slab structure for the convex hull algorithm, and we use Lemma 3.9 to construct appropriate search trees $T_1, \ldots, T_n$ for the $\mathcal{C}$-leaf slabs and for each distribution $\mathcal{D}_i$.

For a $\mathcal{C}$-slab C, we define $\mathrm{seg}(\mathcal{C}, C)$ as the line segment that connects the two vertices of $\mathcal{C}$ that lie on the vertical boundaries of C. Let p be a point outside of $\mathcal{C}$, and let a_1 and a_2 be the vertices of $\mathcal{C}$ in which the two tangents for $\mathcal{C}$ through p touch $\mathcal{C}$. The *pencil slab* for p is the vertical slab bounded by the vertical lines through a_1 and a_2. The *pencil* of p is defined as the region inside the pencil slab for p that lies below the line segments $\overline{a_1 p}$ and $\overline{p a_2}$. A point q is *comparable* to the pencil of p if it lies inside the pencil slab for p. It lies *above* the pencil of p if it is comparable to the pencil of p but not inside it.

6.2 Certificates

We need more refined notions of certificates that relate to the canonical hull $\mathcal{C}$. We remind the reader that a certificate for convex hulls has a sorted list of extremal points in P, and a witness pair for each non-extremal point in P. The points (q, r) form a witness pair for p if $p \in \mathrm{lss}(q, r)$.

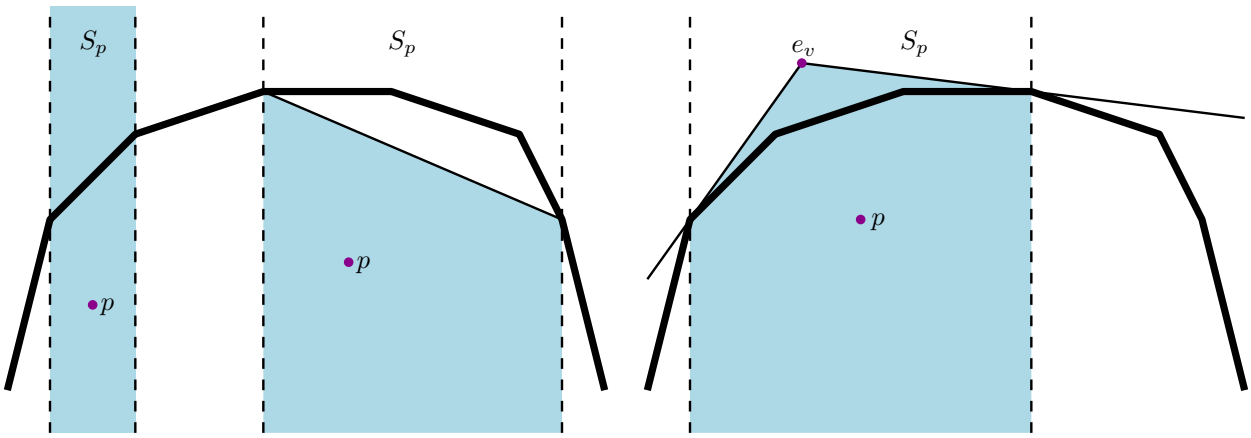

Figure 8: The different possibilities for the $\mathcal{C}$-slab associated with p. The three shaded regions show the three possibilities for S_p. In the left shaded region, p is contained in a leaf slab. The middle region shows a point that is below $\text{seg}(\mathcal{C}, S_o)$. To the right, the point p lies in the pencil of e_v.

A witness pair (q, r) is called *extremal* if both q and r lie on $\text{conv}(P)$. We call (q, r) **V**-*extremal* if both q and r are **V**-extremal. Two distinct extremal points q and r are called *adjacent* if there is no extremal point whose x-coordinate lies strictly between the x-coordinates of q and r. Adjacent **V**-extremal points are defined analogously.

We now define a $\mathcal{C}$-certificate for P. It consists of (i) a list of the **V**-extremal points of P, sorted from left to right; and (ii) a list that has a $\mathcal{C}$-slab S_p for every other point $p \in P$. This $\mathcal{C}$-slab S_p contains p and can be of three different kinds. Either

1. S_p is a $\mathcal{C}$-leaf slab; or
2. p lies below $\text{seg}(\mathcal{C}, S_p)$; or
3. S_p is the pencil slab for a **V**-extremal vertex e_v such that p lies in the pencil of e_v.

The different possibilities are illustrated in Figure 8. The following key lemma is an important piece of the analysis. We relegate the proof to the next section. The reader may wish to skip that section and proceed to learn about the algorithm.

Lemma 6.4. *Assume $\mathcal{C}$ is obtained from a learning phase that succeeds. Let $\mathcal{T}$ be a linear comparison tree that computes the convex hull of P. Then there is an entropy-sensitive linear comparison tree with expected depth $O(n + d_{\mathcal{T}})$ that computes $\mathcal{C}$-certificates for P.*

6.3 From regular certificates to C-certificates: proof of Lemma 6.4

The proof proceeds through various intermediate steps that successively transform a regular certificate into a $\mathcal{C}$-certificate, such that each step incurs only linear overhead in expectation. Then it suffices to apply Lemma 3.4 to obtain an entropy-sensitive search tree of comparable depth.

A certificate γ is *extremal* if all witness pairs in γ are extremal. We provide the chain of lemmas needed and give each proof in a different subsection. The following is proved in Section 6.3.1.

Lemma 6.5. *Let $\mathcal{T}$ be a linear comparison tree that computes* $\text{conv}(P)$*. Then there exists a linear comparison tree with expected depth $d_{\mathcal{T}} + O(n)$ that computes an extremal certificate for P.*

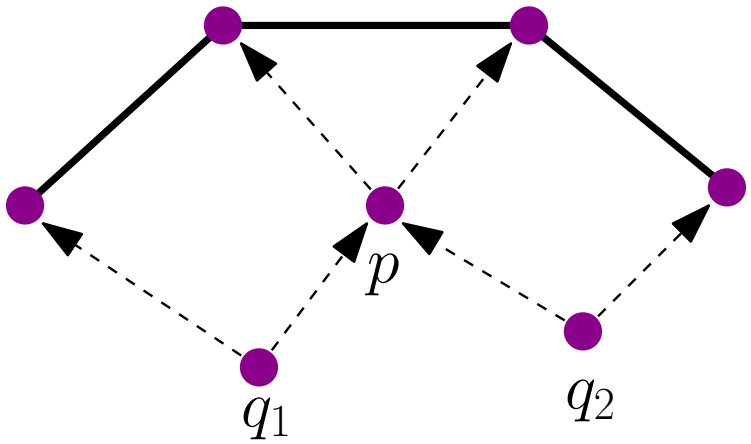

Figure 9: The **shortcut** operation: Observe that computing the convex hull of the out-neighbors of p, q_1, and q_2 suffice for removing p from all witness pairs.

A certificate is $\mathbf{V}$-*extremal* if it contains (i) a list of the $\mathbf{V}$-extremal points of P, sorted from left to right; and (ii) a list that stores for every other point $p \in P$ *either* a $\mathbf{V}$-extremal witness pair for p *or* two adjacent $\mathbf{V}$-extremal points e_1 and e_2 such that $x(e_1) \leq x(p) \leq x(e_2)$. The next lemma is proved in Section 6.3.2.

Lemma 6.6. *Let $\mathcal{T}$ be a linear comparison tree that computes extremal certificates. Then there is a linear comparison tree with expected depth $d_{\mathcal{T}} + O(n)$ that computes $\mathbf{V}$-extremal certificates.*

The final lemma takes us from $\mathbf{V}$-extremal certificates to canonical certificates. The proof is in Section 6.3.3.

Lemma 6.7. *Let $\mathcal{T}$ be a linear comparison tree that computes $\mathbf{V}$-extremal certificates. Then there is a linear comparison tree with expected depth $d_{\mathcal{T}} + O(n)$ that computes $\mathcal{C}$-certificates.*

Lemma 6.4 follows by combining the above lemmas with Lemma 3.4.

6.3.1 Extremal certificates

Proof of Lemma 6.5. We transform $\mathcal{T}$ into a tree that computes extremal certificates. Since each leaf v of $\mathcal{T}$ corresponds to a certificate that is valid for all P with $v = v(P)$, it suffices to show how to convert a given certificate γ for P to an extremal certificate by performing $O(n)$ additional comparisons on P. We describe an algorithm for this task.

The algorithm uses two data structures: (i) a directed graph G whose vertices are a subset of P; and (ii) a stack S. Initially, S is empty and G has a vertex for every $p \in P$. For each non-extremal point $p \in P$, we add two directed edges pq and pr to G, where (q, r) is the witness pair for p according to γ. In each step, the algorithm performs one of the following operations, until G has no more edges left (we will use the terms *point* and *vertex* interchangeably, since we always mean some $p \in P$).

- **Prune.** If G has a non-extremal vertex p with indegree zero, we delete p from G (together with its outgoing edges) and push it onto S.

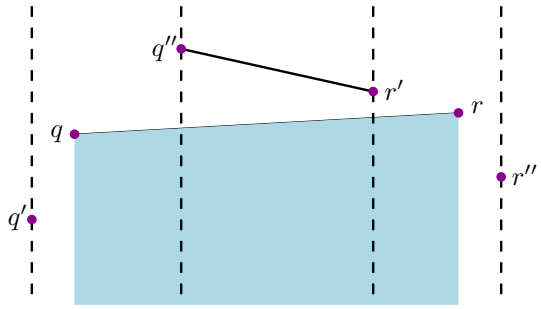

Figure 10: The point p lies in the shaded region. Hence, either $x(p) \in [x(q'), x(q'')]$, p lies in $\mathrm{lss}(q'', r')$, or $x(p) \in [x(r'), x(r'')]$.

- **Shortcut.** If G has a non-extremal vertex p with indegree 1 or 2, we find for each in-neighbor q of p a witness pair that does not include p and we replace the out-edges from q by edges to this new pair. (We explain shortly how to do this.) The indegree of p is now zero.

An easy induction shows that our algorithm maintains the following invariants: (i) all non-extremal vertices in G have out-degree 2; (ii) all extremal vertices of G have out-degree 0; (iii) for each non-extremal vertex p of G, the two out-neighbors of p constitute a witness pair for p; (iv) every $p \in P$ is either in G or in S, but never both; (iv) when a point p is added to S, then we have a witness pair (q, r) for p such that $q, r \notin S$.

We analyze the number of comparisons on P required by each operation. **Prune** needs no comparisons. **Shortcut** is performed as follows: we consider for each in-neighbor q of p the upper convex hull U for p's two out-neighbors and q's other out-neighbor, and we find the edge of U that lies above q. Since the U constant size and since p has in-degree at most 2, this takes $O(1)$ comparisons, see Figure 9. There are at most n **Shortcut**s, so the total number of comparisons is $O(n)$. Note that deciding which operation to perform depends solely on G and requires no comparisons on P.

We now argue that the algorithm cannot get stuck. That means that if G has at least one edge, **Prune** or **Shortcut** can be applied. Suppose that we cannot perform **Prune**. Then each non-extremal vertex has in-degree at least 1. Consider the subgraph G' of G induced by the non-extremal vertices. Since all extremal vertices have out-degree 0, all vertices in G' have in-degree at least 1. The average out-degree in G' is at most 2, so there must be a vertex with in-degree (in G') 1 or 2. This in-degree is the same in G, so **Shortcut** can be applied.

Thus, we can perform **Prune** and **Shortcut** until G has no more edges and all non-extremal points are on the stack S. Now we pop the points from S and find extremal witness pairs for them. Let p be the next point on S. By invariant (iv), there is a witness pair (q, r) for p whose vertices are not on S. Thus, each of q and r is either extremal or we have an extremal witness pair for it. Therefore, we can find an extremal witness pair for p with $O(1)$ comparisons, as in **Shortcut**. We repeat this process until S is empty. This takes $O(n)$ comparisons overall, so we can construct an extremal certificate γ' from γ with $O(n)$ comparisons on P. □

6.3.2 V-Extremal Certificates

Proof of Lemma 6.6. As in the proof of Lemma 6.5, it suffices to show how to convert a given extremal certificate into a **V**-extremal certificate with $O(n)$ comparisons on P. This is done as follows. First, we determine the **V**-extremal points on $\text{conv}(P)$. This takes $O(n)$ comparisons by a simultaneous traversal of $\text{conv}(P)$ and **V**. Without further comparisons, we can now find for each extremal point p in P the two adjacent **V**-extremal points that have p between them. This information is stored in the **V**-extremal certificate.

Now let $p \in P$ be non-extremal, and let (q, r) be the corresponding extremal witness pair. In $O(1)$ comparisons, we will show how to either find a **V**-extremal witness pair, or the right pair of adjacent **V**-extremal points.

We have determined adjacent **V**-extremal points q', q'' such that $x(q) \in [x(q'), x(q'')]$. (If q is itself **V**-extremal, just set $q' = q'' = q$.) Similarly, define adjacent **V**-extremal points r', r''. We know that p lies in $\text{lss}(q, r)$ and hence $x(p) \in [x(q), x(r)]$. Furthermore, the points q', q, q'', r', r, r'' are all in convex position. Since p is in $\text{lss}(q, r)$, one of the following must happen: $x(p) \in [x(q'), x(q'')]$, p lies in $\text{lss}(q'', r')$, or $x(p) \in [x(r'), x(r'')]$; see Figure 10. We can determine which in $O(1)$ comparisons. □

6.3.3 Canonical Certificates

Proof of Lemma 6.7. Similar to the proofs of Lemmas 6.5 and 6.6, we convert a **V**-extremal certificate γ, into a $\mathcal{C}$-certificate with $O(n)$ expected comparisons on P.

This works as follows. The certificate γ provides a list of the **V**-extremal points in P. For each such **V**-extremal point, we perform a binary search to find the $\mathcal{C}$-leaf slab that contains it. This requires $o(n)$ comparisons, since there are at most $n/\log^2 n$ **V**-extremal points and since each binary search needs $O(\log n)$ comparisons. Next, we check, for each $i \leq k$, that the extremal point for v_i lies in ℓ_i^+. This takes one comparison per point. If any of these checks fail, we simply declare failure and use binary search to find for every $p \in P$ a $\mathcal{C}$-leaf slab that contains it.

We now assume that there exists a **V**-extremal point in every ℓ_i^+. (This implies that all **V**-extremal points lie outside $\mathcal{C}$.) We use binary search to determine the pencil of each **V**-extremal point. Again, this takes $o(n)$ comparisons. Now let $p \in P$ be not **V**-extremal. We will use $O(1)$ comparisons and either find the slab S_p or determine that p lies above $\mathcal{C}$. The certificate γ assigns to p two **V**-extremal points e_1 and e_2 such that either (i) (e_1, e_2) is a **V**-extremal witness pair for p; or (ii) e_1 and e_2 are adjacent and $x(e_1) \leq x(p) \leq x(e_2)$. We define f_1 be the rightmost visible point of $\mathcal{C}$ from e_1 and f_2 be the *leftmost* visible point from e_2.

Let us consider the first case. Refer to Figure 11, left. The point p is below $\overline{e_1 e_2}$. Since e_1, f_1, f_2, e_2 are in convex position, $\overline{e_1 e_2}$ is below the convex hull of these points. This means that one of the following must happen: $x(p) \in [x(e_1), x(f_1)]$, $x(p) \in [x(f_2), x(e_2)]$, or p is below $\overline{f_1 f_2}$. This can be determined in $O(1)$ comparisons. In the first two cases, p lies in a pencil (and hence we find an appropriate S_p), and in the last case, we find a witness $\mathcal{C}$-slab. Now for the second case. We will need the following claim.

Claim 6.8. *If, for all i, there exists a **V**-extremal point in ℓ_i^+, then the pencils of two adjacent **V** either overlap or share a slab boundary.*

Proof. Refer again to Figure 11, left. Let e_1 and e_2 be two adjacent **V**-extremal vertices such that their pencil slabs do not overlap or share a boundary. Then f_1 is not visible from e_2. Consider

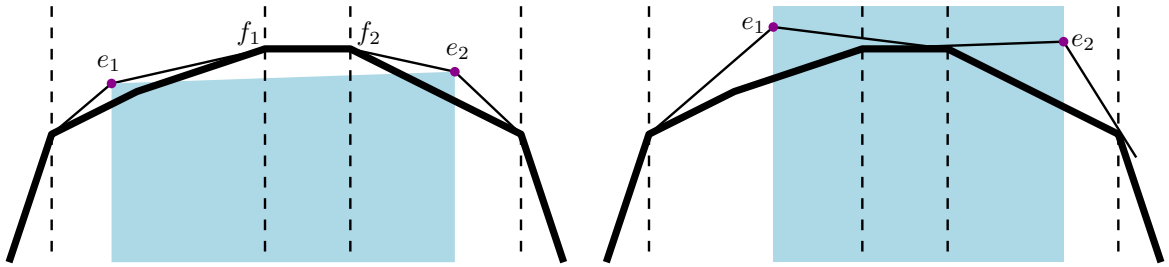

Figure 11: Canonical certificates: In each part, p is contained in the shaded region.

the edge a of $\mathcal{C}$ where f_1 is the left endpoint. The edge a is not visible from either e_1 or e_2 and is between them. By the claim assumption, there exists an extremal point x of P that sees a. But the point x cannot lie to the left of e_1 or to the right of e_2 (that would violate the extremal nature of e_1 or e_2) Hence, x must be between e_1 and e_2, contradicting the fact that they are adjacent. □

With this claim, we can assert that p is comparable to one the pencils of e_1, e_2. By $O(1)$ comparisons, we can check if p is contained in either of these pencils or is above $\mathcal{C}$.

Finally, for all points determined to be above $\mathcal{C}$, we simply use binary search to place them in a $\mathcal{C}$-leaf slab. So for each p, we have found an appropriate S_p and the canonical certificate is complete. We analyze the total number of comparisons. Let X be the indicator random variable for the event that there exist some ℓ_i^+ that does not contain a $\mathbf{V}$-extremal point. Let Y denote the number of points above $\mathcal{C}$. By Corollary 6.2, $\mathbf{E}[X] \leq n^{-3}$ and $\mathbf{E}[Y] = O(n/\log n)$. The number of comparisons is at most $O(Xn \log n + n + Y \log n)$, the expectation of which is $O(n)$. □

6.4 The algorithm

At long last, we have all the tools to describe the details of our convex hull algorithm. It has two parts: the *location algorithm* and the *construction algorithm*. The former algorithm determines the location of the input points with respect to the canonical hull $\mathcal{C}$. It must be careful to learn just the right amount of information about each point, so that the running time is not too large. The latter algorithm uses the resulting information to compute the convex hull of P quickly.

6.4.1 The location algorithm

Using Lemma 3.9, we obtain near-optimal search trees T_i for the slab structure induced by the $\mathcal{C}$-leaf slabs. The algorithm searches progressively for each $p_i \in P$ in its corresponding tree T_i. However, it is important to coordinate the searches carefully and to abort the search for a point p_i as soon as we have obtained enough information about it. The location algorithm maintains the following information.

- **The current slabs C_i.** For each point $p_i \in P$, we store a current $\mathcal{C}$-slab C_i containing p_i that corresponds to a node of T_i.
- **A set of active points A.** The active points are stored in a priority-queue $L(A)$ as in Claim 3.10. The key associated with an active point $p_i \in A$ is the size of the associated current slab C_i (represented as an integer between 1 and k).

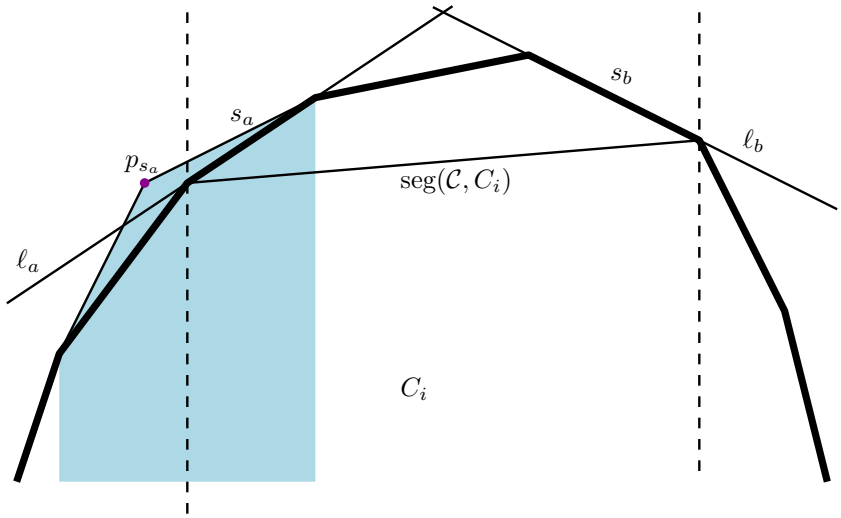

Figure 12: The algorithm: the boundary of C_i is shown dashed, the pencil $\mathrm{pen}(p_{s_a})$ is shaded.

- **The extremal candidates $\tilde{e}_v$.** For each canonical direction $v \in \mathbf{V}$, we store a point $\tilde{e}_v \in P$ that lies outside of $\mathcal{C}$. We call $\tilde{e}_v$ an *extremal candidate* for v.
- **The pencils for the points outside of $\mathcal{C}$.** For each point p that has been located outside of $\mathcal{C}$, we store its pencil $\mathrm{pen}(p)$.
- **The points with the left- and rightmost pencils.** For each edge s of $\mathcal{C}$, we store two points p_{s1} and p_{s2} such that (i) p_{s1} and p_{s2} lie outside of $\mathcal{C}$; (ii) both $\mathrm{pen}(p_{s1})$ and $\mathrm{pen}(p_{s2})$ contain s; (iii) among all pencils seen so far that contain s, the left boundary of $\mathrm{pen}(p_{s1})$ lies furthest to the left; (iv) among all pencils seen so far that contain s, the right boundary of $\mathrm{pen}(p_{s2})$ lies furthest to the right.

Initially, we set $A = P$ and each C_i to the root of the corresponding search tree T_i. The extremal candidates $\tilde{e}_v$ as well as the points p_{s1}, p_{s2} with the left- and rightmost pencils are initialized to the null pointer. Since no point has been located outside of $\mathcal{C}$ so far, there is no pencil $\mathrm{pen}(p)$ yet.

The location algorithm proceeds in *rounds*. In each round, we perform a `find-max` on $L(A)$. Suppose that `find-max` returns p_i. We compare p_i with the vertical line that corresponds to its current node in T_i and advance C_i to the appropriate child. This reduces the size of C_i, so we also need to perform an appropriate `decrease-key` on $L(A)$. Next, we distinguish three cases:
Case 1: p_i lies below $\mathrm{seg}(\mathcal{C}, C_i)$. We declare p_i inactive and `delete` it from $L(A)$.

For the next two cases, we know that p_i lies above $\mathrm{seg}(\mathcal{C}, C_i)$. Let ℓ_a, ℓ_b be the canonical lines that support the edges s_a and s_b of $\mathcal{C}$ that are incident to the boundary vertices of C_i and lie inside of C_i; see Figure 12. We now check where p_i lies with respect to ℓ_a and ℓ_b.
Case 2: p_i is above ℓ_a or above ℓ_b. We now know that p_i lies outside of $\mathcal{C}$. We declare p_i inactive and `delete` it from $L(A)$. Next, we perform a binary search to determine $\mathrm{pen}(p_i)$ and to find all the edges of $\mathcal{C}$ that are visible from p_i. For each such edge s, we compare p_i with the extremal candidate for s, and if p_i is more extreme in the corresponding direction, we update the extremal candidate accordingly. We also update the points p_{s1} and p_{s2} to p_i, if necessary.

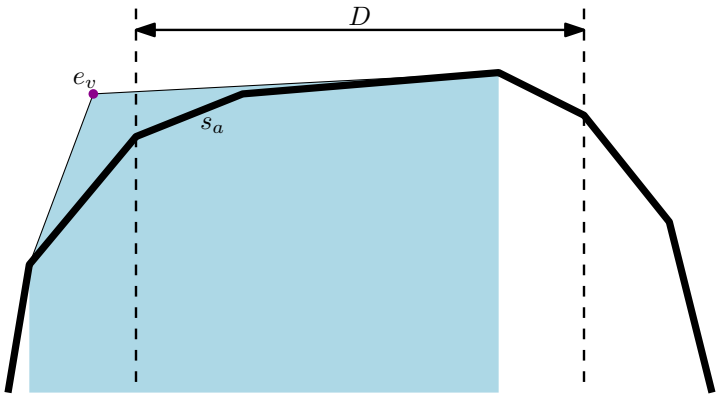

Figure 13: The left boundary of slab D is contained in the pencil slab of e_v.

Case 3: p_i lies below ℓ_a and ℓ_b. Recall that ℓ_a corresponds to the edge s_a of $\mathcal{C}$ and ℓ_b corresponds to the edge s_b of $\mathcal{C}$. We take the rightmost pencil for s_a and the leftmost pencil for s_b (if they exist). This is shown in Figure 12. We compare p_i with these pencils. If p_i lies inside one of the two pencils, we are done. If p_i is above one of the two pencils, we learn that p_i lies outside of $\mathcal{C}$, and we process as in Case 2. In both situations, we declare p_i inactive and `delete` it from $L(A)$. If neither of these happen, p_i remains active.

The location algorithm continues until A is empty (note that every point becomes inactive eventually, because as soon as C_i is a leaf slab, either Case 1 or Case 2 must apply).

6.4.2 Running time of the location algorithm

We now analyze the running time of the location algorithm. We begin with some preliminary claims about the behavior of the algorithm. We note that the algorithm is itself deterministic, and hence we can talk of deterministic properties of the behavior on any input.

Claim 6.9. *Consider the extremal point e_v (in an input P) and let its pencil slab be S. Suppose the search for e_v reaches a slab D such that $|D| \leq |S|$. The algorithm will detect that e_v is an extremal point (for direction v).*

Proof. At least one vertical boundary line of D must be inside (the closure of) S and $D \cap S$ must contain at least one leaf slab. By the definition of a pencil slab, e_v sees all edges of $\mathcal{C}$ in $D \cap S$, so one of the edges s_a or s_b corresponding to D, as defined in the algorithm (refer to Figure 12), must be visible to e_v. Hence, e_v lies in $\ell_a^+ \cup \ell_b^+$ and this is detected in Case 2 of the location algorithm. □

Claim 6.10. *Consider a point p contained in the pencil of an extremal point e_v. Let the pencil slab of e_v be S and suppose the search for p reaches a slab D such that $|D| \leq |S|$. Then, in the next round that p is processed, p becomes inactive.*

Proof. Consider the situation when the search for p has reached D, where $|D| \leq |S|$ and this round completes. The location algorithm schedules the points according to the size of the current slab. Therefore, when p is processed again, all other active points are placed in slabs of size at most $|S|$.

But, by Claim 6.9, if e_v is ever placed in slab of size at most $|S|$, then the algorithm detects that it is extremal and makes it inactive.

Therefore, when p is processed next, e_v has been determined to be the extremal point in the v direction. Note that $D \cap S \neq \emptyset$, since $p \in D \cap S$. Some boundary (suppose it is the left one) of D lies inside S. Let s_a be the corresponding edge of $\mathcal{C}$, as used by the location algorithm; see Figure 13. Since s_a is visible from e_v, and since e_v has been processed already, it follows that the pencil slab of the rightmost pencil for s_a spans all of $D \cap S$. In Case 3 of the location algorithm (in this round), either p will be found inside this pencil slab, or will be found outside $\mathcal{C}$. Either way, p becomes inactive. □

We arrive at the main lemma of this section.

Lemma 6.11. *The total number of rounds in the location algorithm is $O(n + \mathrm{OPT})$.*

Proof. Let $\mathcal{T}$ be an entropy-sensitive linear comparison tree that computes a $\mathcal{C}$-certificate for P in expected time $O(n + \mathrm{OPT})$. Such a tree exists by Lemma 6.4.

Let v be a leaf of $\mathcal{T}$. By Proposition 3.1, we can write $\mathcal{R}_v$ as a Cartesian product $\mathcal{R}_v = \prod_{i=1}^{n} R_i$. The depth of v is $d_v = -\sum_{i=1}^{n} \log \Pr[p_i \in R_i]$, by Proposition 3.3. Now consider a random input P, conditioned on $P \in \mathcal{R}_v$. We will show that expected number of rounds for P is $O(n + d_v)$. This implies the lemma, because the expected number of rounds is

$$\sum_{v \text{ leaf of } \mathcal{T}} \Pr[P \in \mathcal{R}_v] O(n + d_v) = O(n + d_{\mathcal{T}}).$$

Let γ be the $\mathcal{C}$-certificate corresponding to v. The main technical argument is summarized in the following claim.

Claim 6.12. *Consider a point $p_i \in P$, where $P \in \mathcal{R}_v$. The number of rounds involving p_i is at most one more than the number of steps required for an S_{p_i}-restricted search for p_i in T_i.*

Proof. By definition of the canonical certificates, S_{p_i} is of three types. Either S_{p_i} is a $\mathcal{C}$-leaf slab, p_i is below $\mathrm{seg}(S_{p_i}, \mathcal{C})$, or S_{p_i} is a pencil slab of an extremal vertex. In all cases, S_{p_i} contains R_i. When S_{p_i} is a leaf slab, then an S_{p_i}-restricted search for p is just a complete search. Hence, this is always at least the number of rounds involving p_i. Suppose p_i is below $\mathrm{seg}(S_{p_i}, \mathcal{C})$. For any slab $S \subseteq S_{p_i}$, $\mathrm{seg}(S, \mathcal{C})$ is above $\mathrm{seg}(S_{p_i}, \mathcal{C})$. If p_i is located in any slab $S \subseteq S_{p_i}$, it is made inactive (Case 1 of the algorithm).

Now for the last case. The slab S_{p_i} is the pencil slab for a $\mathbf{V}$-extremal vertex e_v, such that the pencil of e_v, $\mathrm{pen}(e_v)$, contains p_i. Suppose the search for p_i leads to slab $D \subseteq S_{p_i}$ and p_i is still active. By Claim 6.10, since $|D| \leq |S_{p_i}|$, p_i becomes inactive in the next round. □

Suppose that P is chosen randomly from $\mathcal{R}_v$. The distribution restricted to p_i is simply random from R_i. By Lemma 3.7, the expected S_{p_i}-restricted search time is $O(1 - \log \Pr[p \in R_i])$. Combining with Claim 6.12, the expected number of rounds is

$$O(n - \sum_{i=1}^{n} \log \Pr[p_i \in R_i]) = O(n + d_v),$$

as claimed. The lemma follows. □

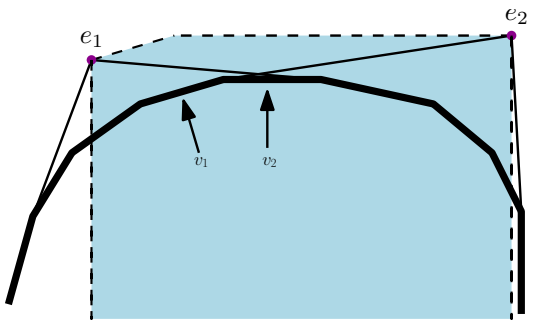

Figure 14: The lines perpendicular to direction v_1 and v_2 define the upper boundary of the shaded region, where p lies. All edges seen by a point in the shaded region can be seen by either e_1 or e_2.

Lemma 6.13. *The expected running time of the location algorithm is $O(n + \mathrm{OPT})$.*

Proof. By Claim 3.10, the total overhead for the data structure operations in linear in the number of rounds. The time to implement Case 1 and Case 3 is $O(1)$, since this just requires comparing p_i with a constant number of lines. Hence, the total time for this is at most the total number of rounds (up to a constant factor).

For Case 2, we perform a binary search for p_i and an extremal point (and pencil) update for every edge s visible from p_i. The binary search only happens if p_i lies outside $\mathcal{C}$. By Corollary 6.3, the expected number of extremal point updates is $O(n/\log n)$. Overall, the total overhead of Case 2 operations is $O(n)$. Combining with Lemma 6.11, the expected running time is $O(n + \mathrm{OPT})$. □

6.4.3 The construction algorithm

We now describe the convex hull construction that uses the information from the location algorithm in order to compute $\mathrm{conv}(P)$ quickly. First, we dive into the geometry of pencils.

Claim 6.14. *Suppose that all $\mathbf{V}$-extremal points of P lie outside of $\mathcal{C}$, and let e_v be a $\mathbf{V}$-extremal point. Then e_v does not lie in the pencil of any other point $p \in P$ outside $\mathcal{C}$.*

Proof. Suppose that $e_v \in \mathrm{pen}(p)$ for another point $p \in P$ that lies outside $\mathcal{C}$. Then a vertex of $\mathrm{pen}(p)$ would be more extremal in direction v than e_v. It cannot be p, since then e_v would not be extremal in direction e_v. But it also cannot be a vertex of $\mathcal{C}$, because e_v lies in ℓ_v^+, while all vertices of $\mathcal{C}$ lie on ℓ_v or in ℓ_v^-. Thus, p cannot exist. □

Claim 6.15. *Suppose that all $\mathbf{V}$-extremal points of P lie outside of $\mathcal{C}$. Let e_1 and e_2 be two adjacent $\mathbf{V}$-extremal points and let $p \in P$ be above $\mathcal{C}$ such that the x-coordinate of p lies between the x-coordinates of e_1 and e_2. Then, the portion of $\mathrm{pen}(p)$ below $\mathcal{C}$ is contained in $\mathrm{pen}(e_1) \cup \mathrm{pen}(e_2)$.*

Proof. By Claim 6.8, the (closures of the) pencil slabs of e_1 and e_2 overlap. Let v_1 be the last canonical direction for which e_1 is extremal and v_2 be the first canonical direction for which e_2 is extremal. Since e_1 and e_2 are adjacent, v_1 and v_2 are consecutive directions in $\mathbf{V}$; see Figure 14. Consider the convex region bounded by the vertical downward ray from e_1, the vertical downward ray from e_2, the line parallel to ℓ_{v_1} through e_1 and the line parallel to ℓ_{v_2} through e_2. By construction, p lies inside this convex region (the shaded area in Figure 14). By convexity, for every

canonical direction $v \in \mathbf{V}$, at least one of e_1 or e_2 is more extremal with respect to v than p. Hence, any edge of $\mathcal{C}$ visible from p is visible by either e_1 or e_2. The portion of $\text{pen}(p)$ below $\mathcal{C}$ consists of the union of regions below edges of $\mathcal{C}$ visible from $\text{pen}(p)$. Therefore, it lies in $\text{pen}(e_1) \cup \text{pen}(e_2)$. □

As described in Section 6.4.1, when the location algorithm is done, for each $p \in P$ we know that either (a) p lies outside of $\mathcal{C}$; (b) p lies inside of $\mathcal{C}$, as witnessed by a segment $\text{seg}(\mathcal{C}, C_p)$; or (c) p lies inside the pencil of a point that we located outside of $\mathcal{C}$. We also have the extremal vertex e_v for all $v \in \mathbf{V}$. We will now show how to use this information in order to find $\text{conv}(P)$. By Corollary 6.3, with probability at least $1 - n^{-2}$, for each canonical direction in $\mathbf{V}$ there is a extremal point outside of $\mathcal{C}$ and that the total number of points outside $\mathcal{C}$ is $O(n/\log n)$. We will henceforth assume these conditions to be true. (If they do not hold, we can compute $\text{conv}(P)$ in $O(n \log n)$. This affects the expected running time by a lower order term.)

For any point a, the ***V*-pair for a** is the pair of adjacent $\mathbf{V}$-extremal points such that a lies between them. The construction algorithm involves a series of steps. The exact details of some of these steps will be given in subsequent claims.

1. Compute the convex hull of the $\mathbf{V}$-extremal points.
2. For each vertex a of $\mathcal{C}$, compute the $\mathbf{V}$-pair for a.
3. For each input point p outside $\mathcal{C}$, compute its $\mathbf{V}$-pair by binary search.
4. For each input point p below a segment $\text{seg}(\mathcal{C}, C_p)$, in $O(1)$ time, either find its $\mathbf{V}$-pair or find a segment between $\mathbf{V}$-extremal points above it. (Details in Claim 6.18.)
5. For each input point p located in the pencil of a non-$\mathbf{V}$-extremal point, in $O(1)$ time, either locate p in the pencil of a $\mathbf{V}$-extremal point or determine that it is outside $\mathcal{C}$. In the latter case, use binary search to find its $\mathbf{V}$-pair.
6. For each input point p located in the pencil of an $\mathbf{V}$-extremal point, in $O(1)$ time, find a segment between $\mathbf{V}$-extremal points above it or find its $\mathbf{V}$-pair. (Details for both steps in Claim 6.19.)
7. By now, for every non-$\mathbf{V}$-extremal $p \in P$, we have found a $\mathbf{V}$-pair or proved it is non-extremal by providing a $\mathbf{V}$-extremal segment about it. For every pair (e_1, e_2) of adjacent $\mathbf{V}$-extremal points, determine the set Q of points that lie above $\overline{e_1 e_2}$. We then take an output-sensitive convex hull algorithm [20] to find the convex hull of Q. Finally, we concatenate the resulting convex hulls to obtain $\text{conv}(P)$.

Claim 6.16. *Once the location algorithm has finished, for each canonical direction $v \in V$, the extremal candidate $\tilde{e}_v$ is the actual extremal point e_v in direction v.*

Proof. By Claim 6.14, e_v does not lie in the pencil of any other point $p \in P$. Hence, the location algorithm must have classified e_v as the extremal candidate for v at some point, and this choice does not change in the further execution of the algorithm. □

Claim 6.17. *The total running time for Steps 1,2,3 and all binary searches in Step 5 is $O(n)$.*

Proof. There are $k = n/\log^2 n$ $\mathbf{V}$-extremal points, so computing their convex hull takes $O(n)$ time. We simultaneously traverse $\mathcal{C}$ and the convex hull of the $\mathbf{V}$-extremal points to determine $\mathbf{V}$-pairs for all vertices of $\mathcal{C}$. We assumed that there are $O(n/\log n)$ points outside $\mathcal{C}$, so that total time for binary searches is $O(n)$. □

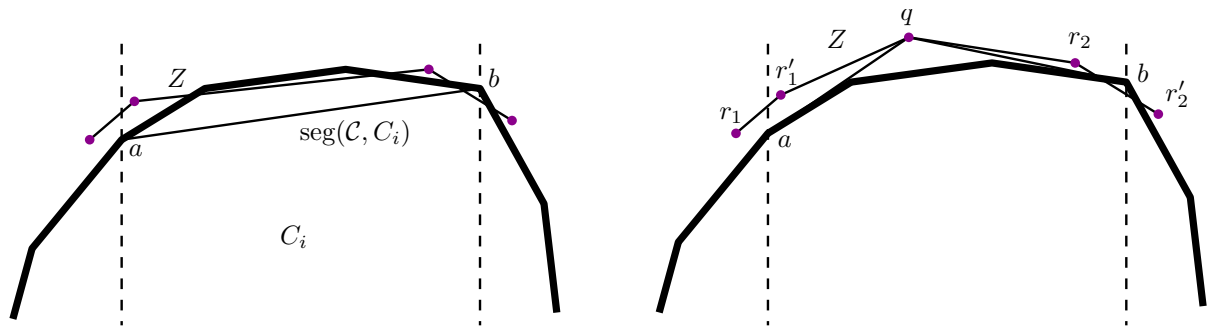

Figure 15: Left: In Step 4, the region below $\text{seg}(\mathcal{C}, C_i)$ is either below the middle segment of Z or between one of the two $\mathbf{V}$-pairs. Right: In Step 6, $\text{pen}(q)$ can be partitioned into regions below $\overline{qr'_1}$, below $\overline{qr_2}$, between r_1, r'_1, or between r_2, r'_2.

Claim 6.18. *Suppose input point p lies below a segment $\text{seg}(\mathcal{C}, C_p)$. Using the information gathered before Step 4, we can either find its $\mathbf{V}$-pair or a segment between $\mathbf{V}$-extremal points above it in $O(1)$ time.*

Proof. Let a and b be the endpoints of $\text{seg}(\mathcal{C}, C_p)$. Consider the convex hull Z of the at most four $\mathbf{V}$-extremal points that define the $\mathbf{V}$-pairs of a and b; see Figure 15, left. The hull Z has at most three edges, and only the middle one (if it exists) might not be between two adjacent $\mathbf{V}$-extremal points. If the middle edge of Z exists, it must lie strictly above $\text{seg}(\mathcal{C}, C_p)$. This is because the endpoints of the middle edge have x-coordinates between $x(a)$ and $x(b)$ and lie outside of $\mathcal{C}$ (since they are $\mathbf{V}$-extremal), while $\text{seg}(\mathcal{C}, C_p)$ is inside $\mathcal{C}$. Now we compare p with the convex hull Z. This either finds a $\mathbf{V}$-pair for p (if p lies in the interval corresponding to the leftmost or rightmost edge of Z) or shows that p lies below a segment between two $\mathbf{V}$-extremal points (if it lies in the interval corresponding to the middle edge of Z). □

Claim 6.19. *Suppose input point p is contained in $\text{pen}(q)$, where q is above $\mathcal{C}$, and the construction algorithm has completed Step 4. If q is non-$\mathbf{V}$-extremal, in $O(1)$ time, we can either find a $\mathbf{V}$-extremal point q' such that $p \in \text{pen}(q')$, or determine that p is above $\mathcal{C}$. If q is $\mathbf{V}$-extremal, then in $O(1)$ time we can find a $\mathbf{V}$-segment above p or find the $\mathbf{V}$-pair for p.*

Proof. Suppose q is not $\mathbf{V}$-extremal. Since q lies outside of $\mathcal{C}$, we already know the $\mathbf{V}$-pair $\{e_1, e_2\}$ for q. By Claim 6.15, if p is below $\mathcal{C}$, then it is either in $\text{pen}(e_1)$ or $\text{pen}(e_2)$. We can determine which (if at all) in $O(1)$ time.

Suppose q is $\mathbf{V}$-extremal. Let a and b be the points of $\mathcal{C}$ on the boundary of $\text{pen}(q)$, where a is to the left: see Figure 15, right. Let (r_1, r'_1) be a's $\mathbf{V}$-pair, where r_1 is to the left. Similarly, (r_2, r'_2) is b's $\mathbf{V}$-pair. The segments $\overline{qr'_1}$ and $\overline{qr_2}$ are above $\text{pen}(q)$. Furthermore, the pencil slab of q is between r_1 and r'_2. One of the following must be true for any point in $\text{pen}(q)$: it is below $\overline{qr'_1}$, below $\overline{qr_2}$, between (r_1, r'_1), or between (r_2, r'_2). This can be determined in $O(1)$ time. □

We are now armed with all the facts to bound the running time.

Lemma 6.20. *With the information from the location algorithm, $\text{conv}(P)$ can be computed in expected time $O(n \log\log n)$.*

Proof. By Claims 6.17, 6.18, and 6.19, the total running time of the first six steps is $O(n)$. Let the $\mathbf{V}$-extremal points be ordered $e_1, e_2, \ldots$. Let X_i denote the number of points in $\text{uss}(e_i, e_{i+1})$ and Y_i be the number of extremal points in this set. Since we use output-sensitive algorithms to compute the hulls of these sets, the running time of Step 7 is $O(\sum_{i \le k} X_i \log(Y_i + 1))$. By Lemma 6.1, this is $O(n \log \log n)$, as desired. □

7 Restricted searches

In this section, we prove Lemma 3.7.

Proof of Lemma 3.7. We bound the expected number of visited nodes in an S-restricted search. Let v be a node of T. In the following, we will use q_v and s_v as a shorthand for the values q_{S_v} and s_{S_v}. Let $\text{vis}(v)$ be the expected number of nodes visited below v, conditioned on v being visited. We will prove below, by induction on the height of v, that for all visited nodes v with $q_v \le 1/2$,

$$\text{vis}(v) \le c_1 + c \log(q_v / s_v), \tag{1}$$

for some constants $c, c_1 > 0$.

Given (1), the lemma follows easily: since T is μ-reducing, it follows that for v at depth k, we have $q_v \le \mu^k$. Hence, we have $q_v \le 1/2$ for all but the root and at most $1/\log(1/\mu)$ nodes below it (note that at each level of T there can be at most one node with $q_v > 1/2$). Let W be the set of nodes w of T such that $q_w \le 1/2$, but $q_{w'} > 1/2$, for the parent w' of w. Since T has bounded degree, $|W| = O(1/\log(1/\mu))$. The expected number $\text{vis}(T)$ of nodes visited in an S-restricted search is at most

$$\begin{aligned}
\text{vis}(T) &\le 1/\log(1/\mu) + \sum_{w \in W} \Pr_{\mathcal{F}_S}[j \in S_w] \text{vis}(w) \\
&\le 1/\log(1/\mu) + c_1 + c \sum_{w \in W} \Pr_{\mathcal{F}_S}[j \in S_w] \log(q_w / s_w) \\
&\le 1/\log(1/\mu) + c_1 + c \sum_{w \in W} \Pr_{\mathcal{F}_S}[j \in S_w] \log(1/s_w),
\end{aligned}$$

using (1) and $q_w \le 1$. By definition of $\mathcal{F}_S$, we have $\Pr_{\mathcal{F}_S}[j \in S_w] = s_w / s_S$, so

$$\begin{aligned}
\text{vis}(T) &\le 1/\log(1/\mu) + c_1 + c \sum_{w \in W} \frac{s_w}{s_S} \log(1/s_w) \\
&= 1/\log(1/\mu) + c_1 + c \sum_{w \in W} \frac{s_w}{s_S} (\log(s_S/s_w) - \log s_S).
\end{aligned}$$

The sum $\sum_{w \in W} (s_w/s_S) \log(s_S/s_w)$ represents the entropy of a distribution over W. Hence, it is bounded by $\log |W|$. Furthermore, $\sum_{w \in W} s_w \le s_S$, so

$$\text{vis}(T) \le 1/\log(1/\mu) + c_1 + \log |W| - c \log s_S = O(1 - \log s_S),$$

as desired.

It remains to prove (1). For this, we examine all possible paths down T that an S-restricted search can lead to. It will be helpful to consider the possible ways that S can intersect the intervals

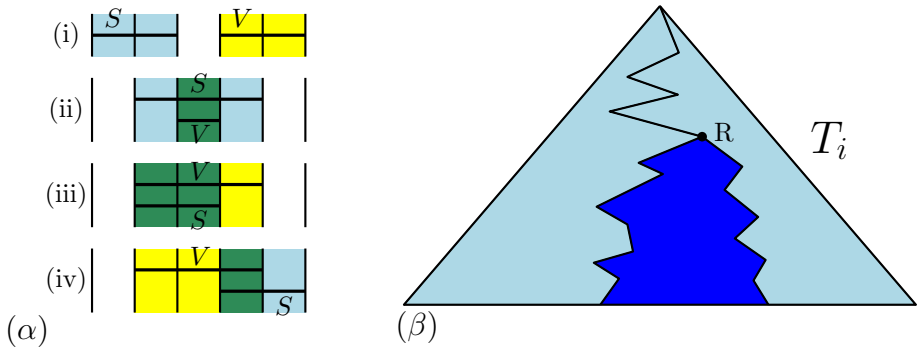

Figure 16: (α) The intersections $S \cap S_v$ in (i)-(iii) are trivial, the intersections in (iii) and (iv) are anchored; (β) every node of T_i has at most one non-trivial child, except for r.

corresponding to the nodes that are visited in a search. Say that the intersection $S \cap S_v$ of S with interval S_v is *trivial* if it is either empty, S, or S_v. Say that it is *anchored* if it shares at least one boundary line with S. Suppose $S \cap S_v = S_v$. Then the search will terminate at v, since we have certified that $j \in S$. Suppose $S \cap S_v = S$, so S is contained in S_v. There can be at most one child of v that contains S. If such a child exists, then the search will simply continue to this child. If not, then all possible children (to which the search can proceed to) are anchored. The search can possibly continue to any child, at most two of which are internal nodes. Suppose S_v is anchored. Then at most one child of v can be anchored with S. Any other child that intersects S must be contained in it. Refer to Figure 16.

Consider the set of all possible nodes that can be visited by an S-restricted search (remove all nodes that are terminal, i.e., completely contained in S). These form a set of paths, that form some subtree of S. In this subtree, there is only one possible node that has two children. This comes from some node r that contains S and has two anchored (non-leaf) children. Every other node of this subtree has a single child. Again, refer to Figure 16. We now prove two lemmas.

Claim 7.1. *Let $v \neq r$ be a non-terminal node that can be visited by an S-restricted search, and let w be the unique non-terminal child of v. Suppose $q_v \leq 1/2$ and $\mathrm{vis}(w) \leq c_1 + c\log(q_w/s_w)$. Then, for $c \geq c_1/\log(1/\mu)$, we have*

$$\mathrm{vis}(v) \leq 1 + c\log(q_v/s_v). \tag{2}$$

Proof. From the fact that when a search for j shows that it is contained in a node contained in S, the S-restricted search is complete, it follows that

$$\mathrm{vis}(v) \leq 1 + \frac{\Pr_{\mathcal{F}_S}[j \in S_w]}{\Pr_{\mathcal{F}_S}[j \in S_v]}\mathrm{vis}(w) = 1 + \frac{s_w}{s_v}\mathrm{vis}(w). \tag{3}$$

Using the hypothesis, if follows that

$$\mathrm{vis}(v) \leq 1 + \frac{s_w}{s_v}(c_1 + c\log(q_w/s_w)).$$

Since $q_w \leq \mu q_v$, and letting $\beta := s_w/s_v \leq 1$, this is

$$\begin{aligned} &\leq 1 + \beta(c_1 + c\log(q_v/s_w) + c\log\mu) \\ &= 1 + \beta c_1 + \beta c\log q_v + \beta c\log(1/s_w) + \beta c\log\mu. \end{aligned}$$

The function $x \mapsto x \log(1/x)$ is increasing in the range $x \in (0, 1/2)$, so $s_w \log(1/s_w) \le s_v \log(1/s_v)$ for $s_v \le q_v \le 1/2$. Together with $\beta = s_w/s_v \le 1$, this implies

$$\begin{aligned} \mathrm{vis}(v) &\le 1 + \beta c_1 + c \log q_v + c \log(1/s_v) + \beta c \log \mu \\ &= 1 + c \log(q_v/s_v) + \beta(c_1 + c \log \mu) \\ &\le 1 + c \log(q_v/s_v), \end{aligned}$$

for $c \ge c_1/\log(1/\mu)$. □

Only a slightly weaker statement can be made for the node r having two nontrivial intersections at child nodes r_1 and r_2.

Claim 7.2. *Let r be as above, and let r_1, r_2 be the two non-terminal children of r. Suppose that $\mathrm{vis}(r_i) \le c_1 + c \log(q_{r_i}/s_{r_i})$, for $i = 1, 2$. Then, for $c \ge c_1/\log(1/\mu)$, we have*

$$\mathrm{vis}(r) \le 1 + c \log(q_r/s_r) + c.$$

Proof. Similar to (3), we get

$$\mathrm{vis}(r) \le 1 + \frac{s_{r_1}}{s_r}\mathrm{vis}(r_1) + \frac{s_{r_2}}{s_r}\mathrm{vis}(r_2).$$

Applying the hypothesis, we conclude

$$\mathrm{vis}(r) \le 1 + \sum_{i=1}^{2} \frac{s_{r_i}}{s_r}[c_1 + c \log(q_{r_i}/s_{r_i})].$$

Setting $\beta := (s_{r_1} + s_{r_2})/s_r$ and using $q_{r_i} \le \mu q_r$, we get

$$\mathrm{vis}(r) \le 1 + \beta c_1 + \beta c \log \mu + \beta c \log q_r + c \sum_{i=1}^{2} (s_{r_i}/s_r) \log(1/s_{r_i}).$$

The sum is maximized for $s_{r_1} = s_{r_2} = s_r/2$, so using once again that $\beta \le 1$, it follows that

$$\begin{aligned} \mathrm{vis}(r) &\le 1 + \beta c_1 + \beta c \log \mu + \beta c \log q_r + c \log(2/s_r) \\ &\le 1 + \beta(c_1 + c \log \mu) + c \log(q_r/s_r) + c \log 2 \\ &\le 1 + c \log(q_r/s_r) + c, \end{aligned}$$

for $c \ge c_1/\log(1/\mu)$, as in (2), except for the addition of c. □

Now we use Claims 7.1 and 7.2 to prove (1) by induction. This bound clearly holds for leaves. For the visited nodes below r, we may then inductively take $c_1 = 1$ and $c = 1/\log(1/\mu)$, by Claim 7.1. We can then apply Claim 7.2 for r. For the parent v of r, we may use Claim 7.1 with $c_1 = 1 + 1/\log(1/\mu)$ and $c \ge c_1/\log(1/\mu)$, getting $\mathrm{vis}(v) \le 1 + c \log(q_v/s_v)$. Now repeated application of Claim 7.1 (with the given value of c) gives that this bound also holds for the ancestors of v, at least up until the $1 + 1/\log(1/\mu)$ top nodes. This finishes the proof of (1), and hence the lemma. □

8 Construction of slabs and search trees

Learning the vertical slab structure $\mathbf{S}$ is very similar to learning the V-list in Ailon et al. [2, Lemma 3.2]. We repeat their construction: think of each $\mathcal{D}_i$ as a distribution generating a single number, the x-coordinate of the sampled point. Take the union of the x-coordinates of the first $t = \log n$ inputs P_1, P_2, ..., P_t. Let the sorted list be $-\infty =: x_0, x_1, \ldots, x_{nt}, x_{nt+1} := +\infty$.

Take the n values $x_0, x_t, x_{2t}, \ldots, x_{(n-1)t}$. This is called the *$V$-list*, and it defines the boundaries for $\mathbf{S}$. We repeat the learning phase result of the self-improving sorter [2, Lemma 3.2].

Lemma 8.1. *For $0 \le j \le n$, let $Z_j = \{x_i | v_j \le x_i < v_{j+1}\}$ be the elements with predecessor v_j. With probability at least $1 - n^{-2}$ over the construction of the V-list, we have $\mathbf{E}_{\mathcal{D}}[|Z_j|] = O(1)$ and $\mathbf{E}_{\mathcal{D}}[|Z_j|^2] = O(1)$, for all $0 \le j \le n$.* □

Lemma 3.8 follows immediately from Lemma 8.1 and the fact that sorting the k inputs P_1, P_2, ..., P_k takes $O(n \log^2 n)$ time. After the leaf slabs have been determined, the search trees T_i can be found using essentially the same techniques as before [2, Section 3.2]. The idea is to use $n^\varepsilon \log n$ rounds to find the first $\varepsilon \log n$ levels of T_i, and to use a balanced search tree for searches that need to proceed to a deeper level. This only costs a factor of ε^{-1}. We restate Lemma 3.9 for convenience. The proof is almost the same as that in [2, Section 3.2], but we redo the proof for our setting. (We require the additional property of restricted search optimality.)

Lemma 8.2. *Let $\varepsilon > 0$ be a fixed parameter. In $O(n^\varepsilon)$ rounds and $O(n^{1+\varepsilon})$ time, we can construct search trees T_1, T_2, ..., T_n over $\boldsymbol{S}$ such that the following holds: (i) the trees can be totally represented in $O(n^{1+\varepsilon})$ space; (ii) probability $1 - n^{-3}$ over the construction of the T_is: every T_i is $O(1/\varepsilon)$-optimal for restricted searches over $\mathcal{D}_i$.*

Proof. Let $\delta > 0$ be some sufficiently small constant and c be sufficiently large . For $k = c\delta^{-2} n^\varepsilon \log n$ rounds and each p_i, we record the leaf slab of $\mathbf{S}$ that contains it. We break the proof into smaller claims.

Claim 8.3. *Using k inputs, we can compute estimates $\hat{q}(i, S)$ for each index i and slab S. The following guarantee holds (for all i and S) with probability at least $1 - n^{-3}$ over the choice of the k inputs. If at least $(c/10e\delta^2) \log n$ instances of p_i fell in S, then $\hat{q}(i, S) \in [(1-\delta)q(i, S), (1+\delta)q(i, S)]$.*[2]

Proof. For a slab S, let $N(i, S)$ be the number of times p_i was in S, and let $\hat{q}(i, S) = N(i, S)/k$ be the empirical probability for this event. Note that $\mathbf{E}[N(i, S)] = kq(i, S)$ and is a sum of independent random variables.

If $kq(i, S) < (c/10e\delta^2) \log n$, then $(c/5\delta^2) \log n > 2e\mathbf{E}[N(i, S)]$. By a Chernoff bound [15, Theorem 1.1, Eq. (1.8)], $\Pr[N(i, S) > (c/5\delta^2) \log n] \le 2^{-(c/5\delta^2) \log n} \le n^{-6}$. Hence, with probability at least $1 - n^{-6}$, if $N(i, S) > (c/5\delta^2) \log n$, then $\mathbf{E}[N(i, S)] \ge (c/10e\delta^2) \log n$.

Assume $\mathbf{E}[N(i, S)] \ge (c/10e\delta^2) \log n$. Using multiplicative Chernoff bounds [15, Theorem 1.1, Eq. (1.7)], $\Pr[N(i, S) \notin [(1 - \delta)\mathbf{E}[N(i, S)], (1 + \delta)\mathbf{E}[N(i, S)]]] < 2 \exp(-\delta^2 \mathbf{E}[N(i, S)]/3) < n^{-6}$. The proof is completed by taking a union bound over all i and S. □

Henceforth, assume that the high-probability event of this claim holds. Note that if $(c/10e\delta^2) \log n$ inputs fell in S, then $\hat{q}(i, S) = \Omega(n^{-\varepsilon})$ and $q(i, S) = \Omega(n^{-\varepsilon})$. The tree T_i is constructed recursively. We will first create a partial search tree, where some searches may end in non-leaf slabs (or, in

[2] We remind the reader that this the probability that $p_i \in S$.

other words, leaves of the tree may not be leaf slabs). The root is the just the largest slab. Given a slab S, we describe the creation of the sub-tree of T_i rooted at S. If $N(S) < (c/10e\delta^2)\log n$, then we make S a leaf. Otherwise, we pick a leaf slab λ such that for the slab S_l consisting of all leaf slabs (strictly) to the left of λ and the slab S_r consisting of all leaf slabs (strictly) to the right of λ we have $\hat{q}(i, S_l) \leq (2/3)\hat{q}(i, S)$ and $\hat{q}(i, S_r) \leq (2/3)\hat{q}(i, S)$. We make λ a leaf child of S. Then we recursively create trees for S_l and S_r and attach them as children to S. For any internal node of the tree S, we have $q(i, S) = \Omega(n^{\varepsilon})$, and hence the depth is at most $O(\varepsilon \log n)$. Furthermore, this partial tree is β-reducing (for some constant β). The partial tree T_i is extended to a complete tree in a simple way. From each T_i-leaf that is not a leaf slab, we perform a basic binary search for the leaf slab. This yields a tree T_i of depth at most $(1 + O(\varepsilon)) \log n$. Note that we only need to store the partial T_i tree, and hence the total space is $O(n^{1+\varepsilon})$.

Let us construct, as a thought experiment, a related tree T_i'. Start with the partial T_i. For every leaf that is not a leaf slab, extend it downward using the true probabilities $q(i, S)$. In other words, let us construct the subtree rooted at a new node S in the following manner. We pick a leaf slab λ such that $q(i, S_l) \leq (2/3)q(i, S)$ and $q(i, S_r) \leq (2/3)q(i, S)$ (where S_l and S_r are as defined above). This ensures that T_i' is β-reducing. By Lemma 3.7, T_i' is $O(1)$-optimal for restricted searches over $\mathcal{D}_i$ (we absorb the β into $O(1)$ for convenience).

Claim 8.4. *The tree T_i is $O(1/\varepsilon)$-optimal for restricted searches.*

Proof. Fix a slab S and an S-restricted distribution $\mathcal{D}_S$. Let $q'(i, \lambda)$ (for each leaf slab λ) be the series of values defining $\mathcal{D}_S$. Note that $q'(i, S) \leq q(i, S)$. Suppose $q'(i, S) \leq n^{-\varepsilon/2}$. Then $-\log q'(i, S) \geq \varepsilon(\log n)/2$. Since any search in T_i takes at most $(1 + O(\varepsilon)) \log n$ steps, the search time is at most $O(\varepsilon^{-1}(-\log q'(i, S) + 1))$.

Suppose $q'(i, S) > n^{-\varepsilon/2}$. Consider a single search for some p_i. We will classify this search based on the leaf of the partial tree that is encountered. By the construction of T_i, any leaf S' is either a leaf slab or has the property that $q(i, S') = O(n^{-\varepsilon})$. The search is of *Type 1* if the leaf of the partial tree actually represents a leaf slab (and hence the search terminates). The search is of *Type 2* (resp. *Type 3*) if the leaf of the partial tree is a slab S is an internal node of T_i and the depth is at least (resp. less than) $\varepsilon(\log n)/3$.

When the search is of Type 1, it is identical in both T_i and T_i'. When the search is of Type 2, it takes at least $\varepsilon(\log n)/3$ steps in T_i' and at most (trivially) $(1+O(\varepsilon))(\log n)$ in T_i. Consider Type 3 searches. The total number of leaves (that are not leaf slabs) of the partial tree at depth less than $\varepsilon(\log n)/3$ is at most $n^{\varepsilon/3}$. The total probability mass of $\mathcal{D}_i$ inside such leaves is $O(n^{\varepsilon/3} \times n^{-\varepsilon}) < O(n^{-2\varepsilon/3})$. Since $q'(i, S) > n^{-\varepsilon/2}$, in the restricted distribution $\mathcal{D}_S$, the probability of a Type 3 search is at most $O(n^{-\varepsilon/6})$.

Choose a random $p \sim \mathcal{D}_S$. Let $\mathcal{E}$ denote the event that a Type 3 search occurs. Furthermore, let X_p denote the depth of the search in T_i and X_p' denote the depth in T_i'. When $\mathcal{E}$ does not occur, we have argued that $X_p = O(X_p'/\varepsilon)$. Also, $\Pr(\mathcal{E}) = O(n^{-\varepsilon/6})$. The expected search time is just $\mathbf{E}[X_p]$. By Bayes' rule,

$$\begin{aligned}\mathbf{E}[X_p] = \Pr(\overline{\mathcal{E}})\mathbf{E}_{\overline{\mathcal{E}}}[X_p] + \Pr(\mathcal{E})\mathbf{E}_{\mathcal{E}}[X_p] &\leq O(\varepsilon^{-1}\mathbf{E}_{\overline{\mathcal{E}}}[X_p']) + n^{-\varepsilon/6}(1 + O(\varepsilon)) \log n \\ &= O(\varepsilon^{-1}\mathbf{E}_{\overline{\mathcal{E}}}[X_p'] + 1)\end{aligned}$$

Since $\Pr(\overline{\mathcal{E}}) > 1/2$, $\mathbf{E}_{\overline{\mathcal{E}}}[X_p'] \leq 2\Pr(\overline{\mathcal{E}})\mathbf{E}_{\overline{\mathcal{E}}}[X_p'] \leq 2\mathbf{E}[X_p']$. Combining all the arguments, the expected search time is $O(\varepsilon^{-1}(\mathbf{E}[X_p'] + 1))$. Since T_i' is $O(1)$-optimal for restricted searches, T_i is $O(\varepsilon^{-1})$-optimal. □

□

9 Learning phase for convex hulls

Here we provide the deferred proofs for the lemmas stated in Section 6.1. We begin with some preliminaries about projective duality and an important probabilistic claim about geometric constructions over product distributions.

Consider an input P. As is well known, there is a *dual* set P^* of lines that helps us understand the properties of P. In particular, we use the standard duality along the unit paraboloid that maps a point $p = (x(p), y(p))$ to the line $p^* : y = 2x(p)x - y(p)$ and vice versa. The *lower envelope* of P^* is the pointwise minimum of the n lines $p_1^*, p_2^*, \ldots, p_n^*$, considered as univariate functions. We denote it by $\mathrm{lev}_0(P^*)$. It is a classic fact that there is a one-to-one correspondence between the vertices and edges of $\mathrm{lev}_0(P)$ and the edges and vertices of $\mathrm{conv}(P)$. More generally, for $z = 0, \ldots n$, the *z-level* of P^* is the closure of the set of all points that lie on lines of P^* and that have exactly z lines of P^* strictly below them. The z-level is an x-monotone polygonal curve, and we denote it by $\mathrm{lev}_z(P^*)$. (Refer to Figure 17.) Finally, the *($\leq z$)-level* of P^*, $\mathrm{lev}_{\leq z}(P^*)$, is the set of all points on lines in P^* that are on or below $\mathrm{lev}_z(P^*)$.

Consider the following abstract procedure. Fix some absolute constant b. For any set S of b lines, let $\mathrm{reg}(S)$ be some geometric region defined by the lines $\{\ell_i \mid i \in S\}$. In other words, $\mathrm{reg}(\cdot)$ is a function from sets of lines of size b to geometric regions (which is just some set in $\mathbb{R}^2$). For example, $\mathrm{reg}(\cdot)$ may be a triangle or trapezoid formed by some lines in S. Consider some such region R and a line ℓ. Let $\chi(\ell, R)$ be some boolean function, taking as input a line and a geometric region.

Suppose we generate a random instance $Q^* \sim \mathcal{D}$. We apply some procedure to determine various subsets $S_1, S_2, \ldots$ of b lines from S. These are chosen based on the values of the sums $\sum_{i \leq n} \chi(q_i^*, \mathrm{reg}(S_j))$. Now generate another random instance $P^* \sim \mathcal{D}$. What can be say about the values of $\sum_i \chi(p_i^*, \mathrm{reg}(S_j))$? We expect them to resemble $\sum_i \chi(q_i^*, \mathrm{reg}(S_j))$, but we have to deal with subtle issues of dependencies. In the former case, S_j actually depends on Q^*, while in the latter case it does not. Nonetheless, we can apply concentration inequalities to make statements about $\sum_i \chi(p_i^*, \mathrm{reg}(S_j))$.

Let J be a set of b indices in $[n]$, and set $Q_J^* := \{q_j^* \mid j \in J\}$. The following lemma can be seen as generalization of Lemma 8.1. The proof is quite analogous to the respective one in [2].

Lemma 9.1. *Fix a constant integer $b > 0$. Let $f_l(n), f_u(n)$ be increasing functions such that for all sufficiently large n, $f_u(n) \geq f_l(n) \geq c' b \log n$ (for sufficiently large constant c'). The following holds with probability at least $1 - n^{-4}$ over a random $Q^* \sim \mathcal{D}$. For all index sets J of size b, if $\sum_{i \leq n} \chi(q_i^*, reg(Q_J^*)) \in [f_l(n), f_u(n)]$, then for some absolute constant $\alpha \in (0, 1)$,*

$$\Pr_{P \sim \mathcal{D}}[\sum_{i \leq n} \chi(p_i^*, reg(Q_J^*)) \in [\alpha f_l(n), f_u(n)/\alpha]] \geq 1 - n^{-3}.$$

Proof. Fix an index set J of size b. Condition on the set of lines $S = \{q_j^* | j \in J\}$ being fixed. Note that the distributions $\mathcal{D}_i$, $i \notin J$ remain the same because of independence. Consider the region $\mathrm{reg}(S)$, which is simply some fixed region. Generate a random Q^* conditioned on $Q_J^* = S$. (This means that we just generate random lines q_i^*, for $i \notin J$.) Focus on the sum $\sum_{i \notin J} \chi(q_i^*, \mathrm{reg}(S))$. Note that $|\sum_{i \notin J} \chi(q_i^*, \mathrm{reg}(S)) - \sum_i \chi(q_i^*, \mathrm{reg}(S))| \leq b$. Suppose $\sum_i \chi(q_i^*, \mathrm{reg}(S)) \in [f_l(n), f_u(n)]$,

then $\sum_{i \notin J} \chi(q_i^*, \mathrm{reg}(S)) \in [g_l(n), g_u(n)]$ (where $g_l(n) = f_l(n) - k$ and $g_u(n) = f_u(n) + k$). What can we say about $\sum_{i \notin J} \chi(p_i^*, \mathrm{reg}(S))$, for an independent $P^* \sim \mathcal{D}$? Observe that since $\mathrm{reg}(S)$ is fixed, $\chi(p_i^*, \mathrm{reg}(S))$ and $\chi(q_i^*, \mathrm{reg}(S))$ are identically distributed.

Define (independent) 0-1 random variables $Z_{J,i} = \chi(p_i^*, \mathrm{reg}(S))$, and let $\hat{Z}_J = \sum_{i \notin J} Z_{J,i}$ and $Z_J = \sum_i Z_{J,i}$. We keep the subscript J because the region S uses fixed lines with indices in J. Given that one draw of $\hat{Z}_J$ is in $[g_l(n), g_u(n)]$, we want to give bounds on another draw. (This is basically a Bayesian problem, in that we effectively construct a prior over $\mathbf{E}[\hat{Z}_J]$. Two Chernoff bounds suffice to perform the argument.)

Claim 9.2. *Consider a single draw of $\hat{Z}_J$ and suppose that $\hat{Z}_J \in [g_l(n), g_u(n)]$, With probability at least $1 - n^{-c'b/5}$, $\mathbf{E}[\hat{Z}_J] \in [g_l(n)/6, 2g_u(n)]$.*

Proof. We use the Chernoff bounds [15, Theorem 1.1]. Suppose $\mu := \mathbf{E}[\hat{Z}_J] < g_l(n)/6$. Then $g_l(n) > 2e\mu$. Hence, $\Pr[\hat{Z}_J \geq g_l(n)] < 2^{-g_l(n)} < n^{-c'b/2}$ (noting that $g_l(n) = f_l(n) - k > (c'b/2)\log_2 n$). With probability at least $1 - n^{c'b/2}$, if $\hat{Z}_J \geq g_l(n)$, then $\mathbf{E}[\hat{Z}_J] \geq g_1(n)/6$.

We repeat the argument with a lower tail Chernoff bound. Suppose $\mu > 2g_u(n)$. Then $\Pr[\hat{Z}_J \leq g_u(n)] \leq \Pr[\hat{Z}_J \leq (1 - 1/2)\mu] < e^{-g_u(n)/4} < n^{-c'b/4}$. With probability at least $1 - n^{c'b/4}$, if $\hat{Z}_J \leq g_u(n)$, then $\mathbf{E}[\hat{Z}_J] \leq 2g_u(n)$. A union bound completes the proof. □

For this claim, we conditioned on the set of lines $S = \{q_j^* \mid j \in J\}$ being fixed. But the claim holds regardless of what the conditioning is, and hence is true unconditionally. Therefore, for a fixed J, with probability at least $1 - n^{-c'b/5}$ over $Q^* \sim \mathcal{D}$, if $\hat{Z}_J \in [g_l(n), g_u(n)]$, $\mathbf{E}[\hat{Z}_J] \in [g_l(n)/6, 2g_u(n)]$. Given that $|\hat{Z}_J - Z_J| \leq b$, this implies: if $Z_J \in [f_l(n), f_u(n)]$, $\mathbf{E}[Z_J] \in [f_l(n)/7, 3f_u(n)]$.

There are $O(n^b)$ possible index sets J, so by a union bound the above holds with probability at least $1 - n^{-c'b/6}$ for all J simultaneously. Suppose we chose a Q such that the condition above held (so $\forall J, \mathbf{E}[Z_J] \in [f_l(n)/7, 3f_u(n)]$). Consider drawing $P \sim \mathcal{D}$, so effectively an independent draw of Z_J. Applying upper tail and lower tail Chernoff bounds again, for some sufficiently small constants α, β, $\Pr[Z_J \in [\alpha f_l(n), f_u(n)/\alpha]] > 1 - \exp(-\beta f_l(n)) > 1 - n^{-3}$. □

9.1 Learning the canonical directions

We now describe how to obtain the canonical directions promised in Lemma 6.1. For this, we take a random input $Q^* \sim \mathcal{D}$.

Take the $(\log^4 n)$-level of Q^*, and let H' the upper convex hull of its vertices. Refer to Figure 17.

Claim 9.3. *The hull H' has the following properties:*

1. *The curve H' lies below $lev_{2\log^4 n}(P^*)$.*
2. *Each line of Q^* either supports an edge of H' or intersects it at most twice.*
3. *H' has $O(n)$ vertices.*

Proof. Consider any (internal) point p on H', and let a, b be the vertices of H' such that p is between them in terms of x-coordinate. Any line that goes under p must go under a or b. There are exactly $\log^4 n$ lines under a (and b), since this is on the $(\log^4 n)$-level. Hence, there are at most $2\log^4 n$ points below p.

The second property is a direct consequence of convexity. The third property follows from the second. Every vertex of H' is present on some line of Q^*, and hence there can be at most $2n$ vertices. □

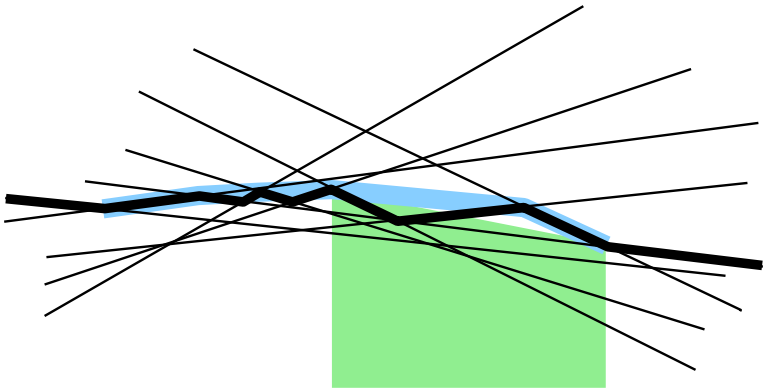

Figure 17: The arrangement of Q^*: for the sake of demonstration, the dark black line is the 4-level of the arrangement. The thick lighter line is H', the convex hull of the vertices in the level. The shaded region is a possible trapezoid τ_j.

Let $r_0, r_1, \ldots, r_k$ be the points given by the every $\log^2 n$-th point in which a line in Q^* meets H' (either as an intersection point or as an endpoint of a segment). For convenience, we will think of these as ordered from right to left. By Claim 9.3(2), there are $O(n/\log^2 n)$ such points r_i. Let H be the upper convex hull of these points r_i. Clearly, H lies below H'. Draw a vertical downward ray through each vertex r_j. This subdivides the region below H into semi-unbounded trapezoids $\tau_0, \tau_1, \ldots$ with the following properties: (i) each vertical boundary ray of a trapezoid τ_j is intersected by at least $\log^4 n$ and at most $2\log^4 n$ lines of Q^* (by Claim 9.3(1)); and (ii) the upper boundary segment of each τ_j is intersected by at most $\log^2 n$ lines in Q^* (by construction). The shaded region in Figure 17 shows one such trapezoid. The next claim follows from an application of Lemma 9.2.

Claim 9.4. *With probability at least $1 - n^{-4}$ (over Q), the following holds for all trapezoids τ_j: generate an independent $P^* \sim \mathcal{D}$.*

1. *With probability (over P^*) at least $1 - n^{-3}$, there exists a line in P^* that intersects both boundary rays of τ_j;*
2. *with probability (over P^*) at least $1 - n^{-3}$, at most $\log^5 n$ lines of P^* intersect τ_j.*

Proof. We apply Lemma 9.2 for both parts. For a set $S = \{\ell_1, \ell_2, \ell_3, \ell_4\}$ of four lines, define $\mathrm{reg}(S)$ as the downward unbounded vertical trapezoid formed by the segment between intersection points of ℓ_1, ℓ_2 and ℓ_3, ℓ_4. Note that all trapezoids τ_j are of this form, where S is a set of four lines from Q^*. Set $\chi(\ell, \tau)$ (for line ℓ and trapezoid τ) to 1 if ℓ intersects both parallel sides of τ and 0 otherwise.

Since τ_j is an unbounded trapezoid, a line that intersects it either intersects the upper segment or intersects both boundary rays. In our sample Q, the number of lines satisfying the former is at most $\log^2 n$ and the number satisfying the latter is in $[\log^4 n, 4\log^4 n]$. Hence, the sum $\sum_{i=1}^{n} \chi(q_i^*, \tau_j)$ is at least $\log^4 n - \log^2 n \geq (1/2)\log^4 n$ and at most $5\log^4 n$. By Lemma 9.2, the number of lines in P^* intersecting both vertical lines is $\Omega(\log^4 n)$ with probability at least $1 - n^{-3}$.

Now for the second part. We redefine $\chi(\ell, \tau)$ to be 1 if ℓ intersects τ and 0 otherwise. Any line that intersects τ_j must intersect one of the vertical boundaries, so $\sum_{i=1}^{n} \chi(q_i^*, \tau_j) \in [\log^4 n, 4\log^4 n]$. By Lemma 9.2, the number of lines in P^* intersecting τ_j is $O(\log^4 n)$ with probability $> 1 - n^{-3}$. □

Proof of Lemma 6.1. Each point r_i is dual to a line r_i^*. We define the directions in $\mathbf{V}$ by taking upward unit normal to the lines r_j^*. (Since the r_i's are ordered from right to left, this gives $\mathbf{V}$ in clockwise order.) It is clear that these directions can be found in $O(n \operatorname{poly}\log(n))$ time: we can compute $\operatorname{lev}_{\log^4 n}(P^*)$ and its convex hull H' in $O(n \operatorname{poly}\log n)$ time [13, 14]. To determine the points r_j, we perform $O(n)$ binary searches over H', and then sort the intersection points. When r_j is known, v_j can be found in constant time.

Now consider a random P and its dual. The $\mathbf{V}$-extremal vertices e_i and e_{i+1} correspond to the lowest line in P^* that intersects the left and the right boundary ray of τ_i. The number of extremal points between e_i and e_{i+1} is the number of vertices on the lower envelope of P^* between $x(r_i)$ and $x(r_{i+1})$. By Claim 9.4(1), this lower envelope lies entirely inside τ_i with probability at least $1 - n^{-3}$. By Claim 9.4(2) (and a union bound), the number Y_i of extremal points between e_i and e_{i+1} is at most $\log^5 n$ with probability at least $1 - 2n^{-3}$. Thus, we have

$$\begin{aligned}
&\mathbf{E}[X_i \log(Y_i + 1)] \\
&\leq \mathbf{E}[X_i \log(\log^5 n + 1) \mid Y_i \leq \log^5 n] \Pr[Y_i \leq \log^5 n] + \mathbf{E}[X_i \log(Y_i + 1) \mid Y_i > \log^5 n] \Pr[Y_i > \log^5 n] \\
&\leq \mathbf{E}[X_i \mid Y_i \leq \log^5 n] \Pr[Y_i \leq \log^5 n] O(\log\log n) + O(n^2)(1/2n^3) \\
&\leq \mathbf{E}[X_i] O(\log\log n) + O(1).
\end{aligned}$$

Adding over i,

$$\begin{aligned}
\sum_{i=1}^{k} \mathbf{E}[X_i \log(Y_i + 1)] \leq \sum_{i=1}^{k} \mathbf{E}[X_i] O(\log\log n) + O(1) = \mathbf{E}\Big[\sum_{i=1}^{k} X_i\Big] O(\log\log n) + O(n) \\
= O(n \log\log n).
\end{aligned}$$

□

9.2 Learning the lines

To compute the canonical lines ℓ_j for the canonical directions $v_j \in \mathbf{V}$, we consider again the dual sample Q^* from the previous section. The jth line ℓ_j is defined as the line that is dual to the point on $\operatorname{lev}_{\gamma c \log n}(Q)$ that has the same x-coordinate as r_j. Here, $\gamma > 0$ is a sufficiently small constant. Note that ℓ_j is normal to v_j. This can be constructed in $O(n \operatorname{poly}(\log n))$ time. We restate the main technical part of Lemma 6.2.

Lemma 9.5. *With probability at least $1 - n^{-4}$ over the construction of $\ell_1, \ell_2, \ldots$, for every ℓ_j,*

$$\Pr_{P \sim \mathcal{D}}[|\ell_j^+ \cap P| \in [1, c \log n]] \geq 1 - n^{-3}.$$

Proof. For convenience, we will go back to dual space. Let s_j be the point that corresponds to ℓ_j. A point $p \in \ell_j^+$ iff p^* intersects that downward vertical ray from s_j (call this R_j)

We set up an application of Lemma 9.2. For a pair of lines ℓ_1, ℓ_2 (all in dual space), define $\operatorname{reg}(\ell_1, \ell_2)$ to be the downward vertical ray from the intersection of ℓ_1 and ℓ_2. Note that any s_j is formed by the intersection of two lines from Q^*. For such a region R and line ℓ', set $\chi(\ell', R)$ to be 1 if ℓ' intersects R and 0 otherwise. Note that, by construction, $\sum_i \chi(q_i^*, R_j) = \gamma c \log n$. We apply Lemma 9.2. With probability at least $1 - n^{-4}$ over Q^* (for sufficiently large c and small enough γ), $\Pr_{P \sim \mathcal{D}}[\sum_i \chi(p_i^*, R_j) \in [1, c \log n]] \geq 1 - n^{-3}$. □

Acknowledgements

C. Seshadhri was supported by the Early Career LDRD program at Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

W. Mulzer was supported in part by DFG grant MU/3501/1.

References

[1] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. In *Proc. 50th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 129–138, 2009.

[2] N. Ailon, B. Chazelle, K. L. Clarkson, D. Liu, W. Mulzer, and C. Seshadhri. Self-improving algorithms. *SIAM J. Comput.*, 40(2):350–375, 2011.

[3] N. Ailon, B. Chazelle, S. Comandur, and D. Liu. Self-improving algorithms. In *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 261–270, 2006.

[4] J. Barbay. Adaptive (analysis of) algorithms for convex hulls and related problems. `http://www.cs.uwaterloo.ca/~jbarbay/Recherche/Publishing/Publications/#asimuht`, 2008.

[5] P. Bose, L. Devroye, K. Douïeb, V. Dujmović, J. King, and P. Morin. Odds-on trees. `arXiv:1002.1092`, 2010.

[6] K. Buchin, M. Löffler, P. Morin, and W. Mulzer. Preprocessing imprecise points for Delaunay triangulation: Simplified and extended. *Algorithmica*, 61(3):674–693, 2011.

[7] C. Buchta. On the average number of maxima in a set of vectors. *Inform. Process. Lett.*, 33(2):63–66, 1989.

[8] B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9(1):145–158, 1993.

[9] K. Clarkson, W. Mulzer, and C. Seshadhri. Self-improving algorithms for convex hulls. In *Proc. 21st Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 1546–1565, 2010.

[10] K. L. Clarkson. New applications of random sampling to computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

[11] K. L. Clarkson and C. Seshadhri. Self-improving algorithms for Delaunay triangulations. In *Proc. 24th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 226–232, 2008.

[12] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4(1):387–421, 1989.

[13] R. Cole, M. Sharir, and C.-K. Yap. On k-hulls and related problems. *SIAM J. Comput.*, 16(1):61–77, 1987.

[14] T. K. Dey. Improved bounds for planar k-sets and related problems. *Discrete Comput. Geom.*, 19(3):373–382, 1998.

[15] D. P. Dubhashi and A. Panconesi. *Concentration of measure for the analysis of randomized algorithms.* Cambridge University Press, Cambridge, 2009.

[16] E. Ezra and W. Mulzer. Convex hull of imprecise points in $o(n \log n)$ time after preprocessing. In *Proc. 27th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 11–20, 2011.

[17] M. J. Golin. A provably fast linear-expected-time maxima-finding algorithm. *Algorithmica*, 11:501–524, 1994. 10.1007/BF01189991.

[18] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 714–723, 1993.

[19] M. Held and J. S. B. Mitchell. Triangulating input-constrained planar point sets. *Inform. Process. Lett.*, 109(1):54–56, 2008.

[20] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299, 1986.

[21] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299, 1986.

[22] M. J. van Kreveld, M. Löffler, and J. S. B. Mitchell. Preprocessing imprecise points and splitting triangulations. *SIAM J. Comput.*, 39(7):2990–3000, 2010.

[23] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[24] M. Löffler and J. Snoeyink. Delaunay triangulation of imprecise points in linear time after preprocessing. *Comput. Geom. Theory Appl.*, 43(3):234–242, 2010.